TURING

图灵程序设计丛书

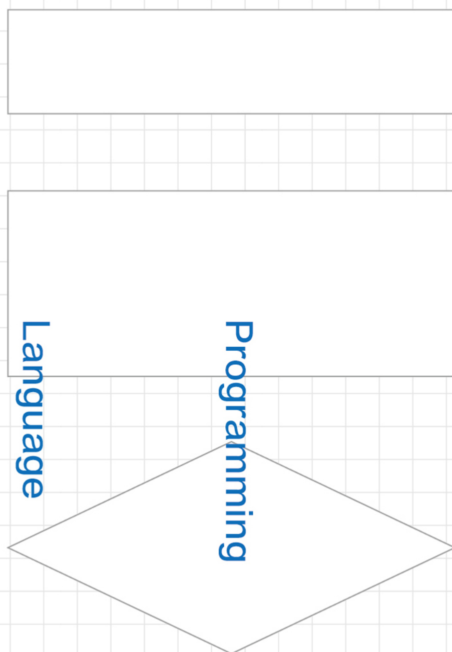
代码之髓

编程语言核心概念

[日] 西尾泰和 著

Nishio Hirokazu

曾一鸣 译



为什么编程语言里会有这些概念？

它们有什么用？应该何时使用？应该如何使用？

运算符 / 循环语句 / 函数 / 异常 / 作用域 / 动态类型 / 类型推断 / 数组 / 字典 / 线程 / 上锁 / 事务

内存 / 类 / 接口 / 继承 / 委托 / C3线性化 / Mix-in / Trait

人民邮电出版社
POSTS & TELECOM PRESS

西尾泰和 | Nishio Hirokazu

24岁取得理学博士学位。2007年起在Cybozu实验室从事提高知识生产力的软件开发工作。曾担任“2011年全日本安全与程序设计实战集训”程序设计语言组组长。特别关注编程语言的多样性及发展。著作有《Jython语言程序设计》《程序员应该了解的程序设计基础知识》《WEB+DB PRESS》（第60期特辑）等。

曾一鸣 | 译者

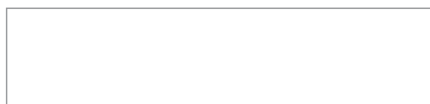
2010年上海交通大学电子工程系研究生毕业，现就职于某国际独立软件开发商，从事软件售后支持工作。对面向对象程序设计、脚本语言及其在语音和图像等信号处理中的应用有着浓厚的兴趣。

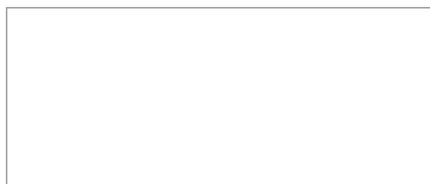
TURING
图灵程序设计丛书

代码之髓

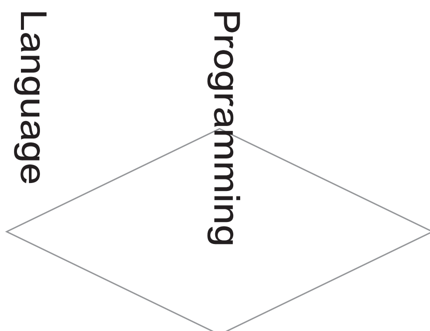
编程语言核心概念

[日] 西尾泰和 Nishio hirokazu 著
曾一鸣 译





Programming Language Concepts



人民邮电出版社
北京

图书在版编目（CIP）数据

代码之髓：编程语言核心概念 /（日）西尾泰和著；
曾一鸣译. -- 北京：人民邮电出版社，2014.8
（图灵程序设计丛书）
ISBN 978-7-115-36153-0

Ⅰ.①代… Ⅱ.①西… ②曾… Ⅲ.①程序语言
Ⅳ.①TP312

中国版本图书馆CIP数据核字（2014）第137918号

内 容 提 要

本书作者从编程语言设计的角度出发，围绕语言中共通或特有的核心概念，通过语言演变过程中的纵向比较和在多门语言中的横向比较，清晰地呈现了程序设计语言中函数、类型、作用域、类、继承等核心知识。本书旨在帮助读者更好地理解各种概念是因何而起，并在此基础上更好地判断为何使用、何时使用及怎样使用。同时，在阅读本书后，读者对今后不断出现的新概念的理解能力也将得到提升。

本书力求简明、通俗，注重可读性，可作为大学计算机科学和软件工程等专业程序设计语言概论教材、计算机等级考试的参考资料，也可作为软件开发人员的学习参考书。

◆ 著　　　　［日］西尾泰和
　译　　　　曾一鸣
　责任编辑　徐　骞
　责任印制　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷
◆ 开本：880×1230　1/32
　印张：7.375
　字数：235千字　　　　2014年8月第1版
　印数：1–4 000册　　　2014年8月北京第1次印刷

著作权合同登记号　图字：01-2013-9190号

定价：45.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第0021号

前言

当今程序设计语言多种多样，可供阅读的资料也非常多。但一个人的学习时间是有限的，全部都学并不现实。

另外，信息技术瞬息万变，特定语言及工具很快便已陈旧。如果不能意识到这一点而有选择性地学习一些相对稳定的知识，所学的内容将逐渐失去价值。

那么，该学习哪些知识并如何学习呢？笔者认为在学习中需要做到以下三点。

- 在比较中学习
- 在历史中学习
- 在实践中学习

第一条是指通过比较多种语言，总结出某种语言的独有特点，以及多种语言的共有特点。

第二条是指通过追溯语言的发展历史，了解语言是如何产生、变化和消失的，探寻语言发展演变的轨迹。

第三条是指亲自进行程序设计。边实践边思考如何编程，才能深入理解语言设计者的意图，同时也能发现自己原先理解不到位之处。

在阅读了各种程序设计书籍之后，相信读者们都曾产生过很多疑问。本书的目的就是解答大家的这些疑惑。本书假设读者对程序设计还不是很熟悉，侧重讲解“在比较中学习”和“在历史中学习”。如果大家在阅读本书后能掌握这些学习方法，那我将不胜欣喜。

致谢

本书在编写过程中受到多位同仁和朋友的大力支持。在此，请允许我省略敬称，对各位的帮助表示最诚挚的谢意（排名不分先后）：和田英一、井出真广、佐藤敏纪、长慎也、鹫见正人、石本敦夫、泷泽昭广、文殊堂（monjudoh）、小泉守义、笹田耕一、园田裕贵（Yugui）、庄司嘉织、尾崎智仁、水岛宏太、矢野勉、松原丰、cactusman、都元daisuke、小田切笃、竹迫良范、光成滋生、中谷秀洋、蓑轮太郎、中山心太、Pawel Marczewski、Harry Roberts。

此外，技术评论社的池田大树先生对本书的出版倾尽心血，在此一并表示感谢。

最后，我要感谢我的妻子，在她的协助下，我才有充分的时间投入到本书的编写当中。

2013 年 3 月　西尾泰和

本书构成

本书共分为 12 章。

第 1 章围绕如何深入高效地学习语言，举例说明“在比较中学习”和“在历史中学习”两种学习方法。

第 2 章探讨程序设计语言是如何产生的。

从第 3 章开始将介绍和程序设计语言相关的各种概念。本书不以某特定语言为叙述前提，如果讲解的一些知识大家尚未接触到，理解起来可能有些困难。比如，只有 C 语言相关经验的读者可能对第 6 章错误处理的内容理解起来稍显吃力。如果没有使用过线程，就不太好理解第 10 章的并行处理。这时，大家可以先阅读其他章节。

第 3 章重点讨论程序设计语言中为什么有那么多的语法规则。着重介绍运算符的优先顺序，并比较规则相对较少的 FORTH 语言和 LISP 语言。

第 4 章通过比较不具有控制语句的汇编语言和具有控制语句的 C 语言，探讨 if、while、for 等控制语句产生的原因。

第 5 章讨论函数是如何产生的，并学习递归调用的使用场合。

第 6 章围绕现在许多语言中称为异常的错误处理机制，介绍这一机制的必要性以及它的发展过程。

第 7 章阐述变量与函数的名字产生的原因，并介绍作用域的必要性及其进化的过程。

第 8 章探讨类型存在的必要性。首先从数的表达方法讲起，接着介绍类型及其应用。

第 9 章会学习一种能存入多个元素的物体，即容器。容器种类多样，本章将解释为什么会有这么多种类、各种类型之间的差异，以及各自的优缺点。本章后半部分会学习字符串，在历史中学习字符编码，并在比较中学习不同语言中字符串的差异。

第 10 章讨论的是同时执行多个处理的并行处理中存在的问题以及它的规避策略，我们将在不同语言的比较中展开这部分内容。

第 11 章将学习面向对象。首先，通过比较 SmallTalk 语言和 C++ 语言，介绍面向对象这一术语指示的内容在不同语言中有哪些差异。接下来将阐述面向对象发明的原因。另外，本章还会介绍类和其他不同的对象创建机制。

第 12 章围绕继承，探讨不同语言中的继承机制及它们各自的优缺点。

示例代码下载

本书中的示例代码，以及在编写本书过程中供验证使用的代码，均可从笔者创建的支持网站上下载。但因篇幅所限，部分代码没有包含在本书中。

http://nhiro.org/langbook/

本书是在弊社杂志《WEB+DB PRESS》Vol.66 特辑之《程序员应该知道的程序设计基础知识——熟知语言核心设计如有神助》(プログラマが知るべき言語設計の基礎知識——言語の核を知れば､自ずと作法が見えてくる)的基础上，大量添加内容并修正之后编纂而成。

本书展示的代码已在 Mac OS 10.7.5 的以下程序语言环境中运行并确认过。不同的执行环境可能带来操作顺序、显示画面及执行结果的差异。

- Scala: version 2.9.2 (Java HotSpot 64-Bit Server VM, Java 1.7.0_05)
- Haskell: GHCi, version 6.10.4
- Python: 2.7.3
- Ruby: 1.9.3p194 (2012-04-20 revision 35410) [x86_64-darwin11]
- Perl: v5.12.4 built for darwin-thread-multi-2level
- JavaScript: Node.js.v0.6.18
- Java: build 1.7.0_05-b05
- C/C++: gcc version 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.9.00)

目录

第3章

第4章

第6章

第1章 如何深入高效地学习语言

“内容能够理解，但总觉得不够透彻。”

大家在学习编程的过程中有过这种感觉吗?

当新学的知识与自身经验以及原来掌握的知识尚未很好结合的时候，往往会出现这种似懂非懂的状态。

“要学的东西太多了，先学什么好呢?”

大家曾为这种问题苦恼过吗?

我们都想集中精力学习一些知识要点，但是怎样才能做到呢?

1.1 在比较中学习

假设你正在学习一种编程语言 X，并为区分知识要点和非要点而苦恼。这时，如果你开始学习另一种编程语言 Y，这个问题可能就会迎刃而解。因为你开始了解那些因语言不同导致的差异，什么规则是 X 和 Y 共通的，什么又是 X 语言独有的。

多种语言共通的知识才是要点。掌握了这些要点，学习其他语言时才会更加轻松。

语言不同，规则不同

在比较中学习多种语言时，一些知识能理解得更深刻。所谓语言不同，规则不同。

编程语言的教材中会罗列出各种各样的规则。其实这些规则并不具有普遍意义，只是因为“在当前的特定情况下，做此规定能更方便”[①]。

① “这样写更自然，那就规定这样写吧。规定这东西唾手可得。”这句名言出现在竹内郁雄所著的《初めての人のための LISP[増補改訂版]》(翔泳社，2010 年出版。中文译名：LISP 基础教程）一书中。

某种语言的教材里出现的某某规则不过是该语言里的规则，仅此而已。

C 语言和 Ruby 语言中的真假值

我们来看一下决定孰真孰假的真假值。学过 C 语言的人都被告知 0 是假、其余为真。于是仅仅学过 C 语言的人就容易误解为在程序设计中一般 0 就代表假、其余为真。因此，等到开始学习 Ruby 语言，发现在 Ruby 中 0 是真时，不免十分惊讶。

C语言中0是假，所以显示为"false!"

```
#include <stdio.h>

int main(){
  if(0){
    printf("true!\n");
  }else{
    printf("false!\n");
  }
}
```

Ruby语言中0是真，所以显示为"true!"

```
if 0 then
  print "true!"
else
  print "false!"
end
```

我们可以借助这次恍然大悟的机会，来修正由来已久的错误想法。即，并不是一般情况下 0 都为假、其余为真，在 C 语言中，0 为假其余为真，而在 Ruby 中，false 和 nil 为假其余（包括 0 在内）都为真。

那其他语言又是什么情况，大家有兴趣了解吗？笔者是颇有兴趣的。这时，笔者有了一个明确的目的，想知道其他语言中真假是如何定义的。目的明确了，学习效率自然而然就提高了。

Java 语言中的真假值

笔者比较过各种语言，这里只列举其一。Java 语言是有真假值这一

数据类型的，在条件语句中必须使用这种类型。因为 0 为整型而不是真假值类型，如果在条件语句中用 0 作判断条件，就要发生编译错误。可见，0 为真、0 为假、0 既非真亦非假的语言都是存在的。[①]

1.2 在历史中学习

理解语言设计者的意图

设想你在阅读关于编程语言某种功能的介绍时，脑子里总有一种不够透彻的感觉。这时，你想知道为什么需要这种功能。

编程语言也是人创造出来的。知道了语言设计者为解决何种问题而创造了这种语言，以及这种语言经历过怎么样的历史变迁后，慢慢地就能理解为什么需要有这种功能了。

应该学哪种语言，我们无从所知

了解了语言的历史，我们往往更能加深所学。“想学编程，但该学哪种语言呢?”这个问题没有意义。可能有很多人会给出一些建议，比如，学好某某主流语言就可以高枕无忧了；今后这个领域会有大的发展，趁现在赶紧把某某语言学了吧。但是，未来的事情谁也说不准。

我们来看一下某种语言的介绍吧。为了便于说明，我们隐去部分语句，并添加一些补充注释来帮助读者理解。

> （要理解某些语言）我们需要具备相当专业的知识，因此使用起来难免感到力不从心。与此不同的是，作为一种工具，X 这种语言能帮助业务负责人和管理者在短时间内获得所需信息，逐渐受到重

① 再来看一下非数值的情况。Python 语言中 0 为假，大小为 0 的容器也定义为假，所以空字符串与空的列表也为假。C 语言中用于处理字符串的 Char* 即使指向的字符串为空也不为假，而当不指向任何值（值为 NULL）时便为假。

> 视。最近两三年间，在终端用户所在部门普及X语言的企业，从美国迅速扩散到其他各国。在（X语言普及较早的）美国某公司Y中，过去几年内，X语言用户的年增长率超过50%，约25 000名以上员工在日常业务中以不同方式使用X语言，占公司总员工数的16%。（中略）在提高企业生产效率方面，X语言被视为一种越来越有效的手段。

从这段文字描述可知，X语言真是一门相当不错的编程语言。如果现在就有人劝你学习这门语言，你会动心吗？

这篇文章其实是1978年刊登在日本信息处理协会杂志上的一篇相当老的文章[①]。X语言其实就是IBM（前述的Y公司）在1964年发布的APL语言。如今，它的使用需求骤减，退出了主流语言的舞台[②]。被称为C语言圣经的*The C Programming Language*[③]一书问世的时间就是1978年。现今，C语言的使用已经变得十分广泛。

学习适用于各种语言的知识

现在还有很多被不同人以不同理由推荐学习的编程语言。然而，在5年后、10年后，单个语言的知识是否依然有用？没人能说清楚。通过比较不同的语言、了解语言的发展历史及其变化原因，培养对不同语言都适用的理解能力，是非常重要的。

---

① “APL”,《信息处理》，竹下亨，Vol.19 No.1, 1978年

② 在招聘网站Dice.com上检索一下，APL语言的没落程度便可见一斑。2013年招聘要求中，提到APL的信息仅有5条。提到Java语言的有16295条，提到Python语言的有3502条。由此可见，使用APL语言的人已经很少了。

③ 中文版为《C程序设计语言（第2版•新版）》，机械工业出版社于2004年1月出版，徐宝文译。

1.3 小结

本书并不是只介绍某一特定语言，而是着眼于学习具有普遍适用性的知识。为此，我们使用“在比较中学习”和“在历史中学习”这两种方法。

“在比较中学习”不是学习某种特定语言的编程，而指的是同时比较几种语言，从而掌握哪些知识是因语言不同而不同的，哪些知识是几种语言共通的。

“在历史中学习”指的是探寻语言是如何变化的，以及在发生变化前存在哪些问题，从而理解语言为何开发出各种功能。

第2章 程序设计语言诞生史

程序设计语言是如何诞生的?

前人是基于什么目的发明了程序设计语言?

本章我们来回顾一下程序设计语言诞生的历史。

2.1 程序设计语言诞生的历史

在第 1 章中我们讲到，通过比较旧事物和新事物可以加深理解。其实，了解旧事物还有另外一个好处。

很多事物都是在过去的基础上，通过不断积累创造出来的。新事物是在充分了解了旧事物的基础上发展起来的。现在那些看似理所当然的事物在过去可能不为世人所知。因此，对于初学者来说，学会从前人的视角来考虑问题，是十分有益的。

那么，我们赶紧来回顾一下历史吧。程序设计语言是如何产生的呢？创造程序设计语言的目的是什么呢？过去的语言和现今的语言有何共通点呢？

程序设计语言的产生是为了让人们的生活、工作更加便捷。为了说明这一点，我们先来看一下程序设计语言诞生的历史，以及语言设计者的设计思想。

连接电缆

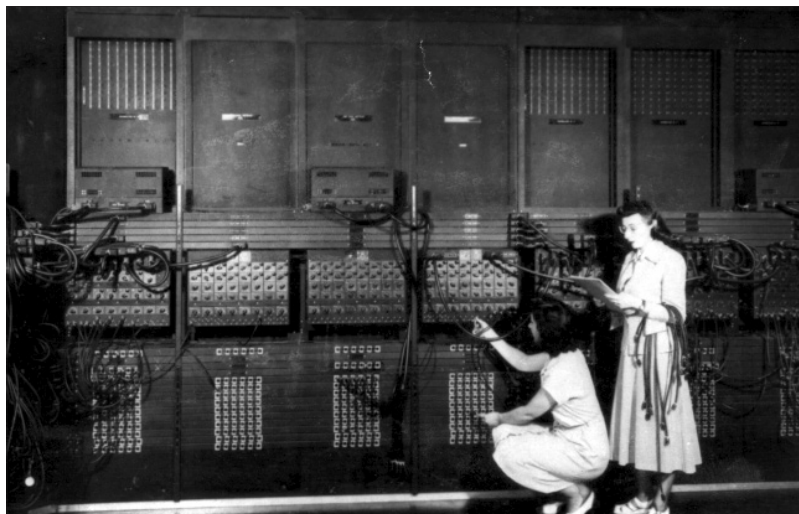
大约半世纪以前，程序设计是个什么概念呢？

1946 年，世界上第一台电子计算机——ENIAC（埃尼阿克，Electronic Numerical Integrator and Computer）问世。它可以改变计算方式，即可以更改程序。用现在的话来讲，它是一台可编程计算机。但是其编程方法和如今大家熟知的程序设计大相径庭。

ENIAC 是一台超大型的计算机，使用了 17 468 个真空管，长达 24 米。试想一下，在某小学的游泳池内，沿 25 米长的泳道，摆放上布满真空管的机械装置，这是怎样的一幅场景！

当时的程序设计就是指把这台计算机不同的端口通过电缆连接起来（图 2.1）。每次更改程序时都要重新调整电缆连接方式，实在费劲。有没有方便一点的方式呢？

图 2.1　ENIAC 的编程场景

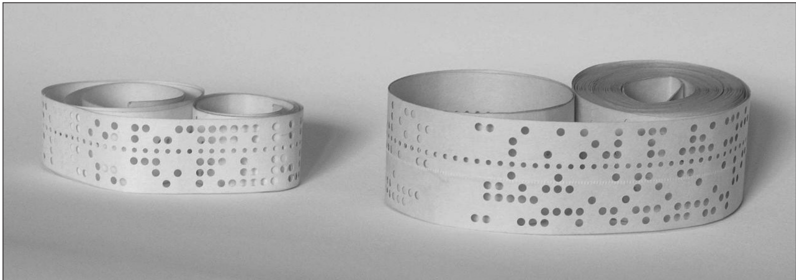


※ U.S. Army Research Laboratory

程序内置

1949 年，EDSAC（爱达赛克，Electronic Delay Storage Automatic Calculator，电子延迟存储自动计算机）问世。这是一种通过纸带打点的方式来记录和读取数据的计算机（图 2.2）。程序作为数据通过纸带输入。不需要重新连接电缆，只需要让计算机不断读取纸带上的数据就可以更改程序。这样一来，程序的更改变得简单易行。

图 2.2　纸带



※ “Puchned tape” by TedColes. 2012 年 1 月 8 日 22 点（日本时间）的最新版。http://en.wikipedia.org/wiki/File:PaperTapes-5and8Hole.jpg

但是人们要读懂这种程序绝非易事。因为这毕竟只是一种供机器（计算机）阅读的语言，即机器语言。在输入纸带上，每列最多有 5 个孔用来记录数据，其他的小孔用于纸带传送。程序只能通过这 5 个孔表现出来。[①]

FORTRAN 语言问世

直到 1954 年，与大家现在使用的语言类似的程序设计语言才被发明出来。这就是 FORTRAN。[②] 它的全称是 Formula Translating System（公式翻译系统）。现在，我们常用 X*Y+Z 来表达“X 乘以 Y 再加 Z”。最早实现这一点的就是 FORTRAN。将公式转化为机器语言是 FORTRAN 语言的特点之一。

在那个年代，人们普遍认为只有用机器语言才能写出高效的程序。[③]

① 读者如有兴趣了解 EDSAC 程序和它的执行过程，可以到笔者设计的基于浏览器的 EDSAC 仿真器上检验一下。http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html

② 这并不是说 FORTRAN 是最早的程序设计语言。哪一种语言最早出现仍然众说纷纭。在此之前就有许多种语言。大多数读者脑海中的程序设计语言应该不是 LISP 语言、FORTH 语言，也不是汇编语言吧。

③ 参见 Ravi Sethi 的著作 *Programming Languages: Concepts and Constructs*。中文版为《程序设计语言：概念和结构（原书第 2 版）》，由机械工业出版社于 2002 年出版，裘宗燕等译。

实际上，使用 FORTRAN 语言编译出来机器语言，与一个熟练的程序设计者直接手写机器语言相比，效率更低。但因 FORTRAN 语言的可读性强且代码编写量大大减少，它还是俘获了众多用户的心。

1979 年，FORTRAN 的设计者 John Backus 说道：我的大部分成果源自我的懒惰[①]。因为我不喜欢写程序，所以我设计出了能轻松编写程序的系统[②]。

2.2 程序设计语言产生的原因

我们为了获得更轻松便捷的体验而编写程序。但轻松便捷不等于偷工减料。偷工减料在前，痛苦在后，这不是真正的便捷。

懒惰：程序员的三大美德之一

大家听说过“程序员的三大美德”吗？ Perl 语言的设计者 Larry Wall 在其著作 *Programming Perl*[③] 中提出，优秀的程序员具有三大美德：懒惰、急躁和傲慢（Laziness, Impatience and Hubris）。这就是俗称的程序员的三大美德。本节，我们介绍其中最重要的一项素质：懒惰。[④]

> 懒惰（Laziness）
>
> 懒惰是一项为了减少总能量支出，而不遗余力地努力的素质。为了节省工夫，设计的程序逐渐被更多的人使用。单独回答每个使用者的疑问费时费力，于是，程序中开始标有注释。所以说，懒惰

① “Much of my work has come from being lazy, ” http://www.msnbc.msn.com/id/17704662/.

② “I didn’t like writing programs,... I started work on a programming system to make it easier to write programs.” http://www.msnbc.msn.com/id/17704662/.

③ 中文版为《Perl 语言编程（第 3 版）》，由中国电力出版社于 2001 年出版，何伟平译。

④ 美德中“急躁”的意思是，程序员忍受不了程序执行的低效。“傲慢”的意思是，程序员容不得对错误不管不顾。

是程序员最宝贵的素质。也正因如此，本书才得以展现在各位面前。请参考急躁和傲慢的解释。

《Perl 语言编程（第 3 版）》

Laziness 有懒惰、懒散、慵懒等不同的翻译方式，总的来说就是让自己轻松、方便。但这不是追求一时轻松，而是选择能将轻松便捷最大化的方法。也就是说，在能达到相同目的的多种方法中，选取一种效率最高、效果最好的方法。

根据《Perl 语言编程（第 3 版）》一书，Perl 这一名字是来自 Practical extraction and report language（实用的数据获取及展示语言）。可见，Perl 是为了能方便地展示数据而发明的一种语言。

语言们各有各的便捷

前面说到程序设计语言是为寻求便捷而创造的。那么，为什么需要有这么多种语言呢？这是因为，大家对于便捷的理解因人而异。我们来看一下语言设计者们的目的以及他们是以何为便捷的吧。

何为“便捷”

语言旨在使什么变得便捷呢？是高速的代码执行？还是简单易于掌握的语言规范？抑或是轻松地理解他人编写的代码？

比如，C++ 是一种非常重视代码执行速度的语言。为了使编程实现相同目的时，执行速度不亚于 C 语言，C++ 语言的规范相应变得复杂了。

另外，Scheme 是一种很重视语言规则是否容易掌握的编程语言。它追求语言规范最简原则，所以它的语言规范全部加起来只有紧凑的 50 页而已①。但是，对满是括号的书写方式存在抵抗情绪的人应该不少。

Python 是一种侧重于把代码阅读变得容易的语言。相对于 Scheme

① 准确来讲，在 1998 年第五版修订版前是这样的。第五版修订版是 50 页，2007 年的第六版放弃了规范最简原则，增加到 187 页。然而，C++ 语言有着超过 1300 页的规则说明书。相比之下，Scheme 语言算是非常紧凑的了。（这里的页数都是指日文版页数。——译者注）

语言，它更接近于 C 语言。熟练的编程人员会使用很多控制语句，并且会在结构层面通过缩进符来规范书写。相对应地，其速度不是特别快，语言规则也不那么少而紧凑。

各有各的便捷

语言的便捷之处各不相同。比如，用 PHP 语言编写 Web 服务很轻松，但它不擅长文字处理。相反，Haskell 和 OCaml 这样的 ML（Meta-Language）系列语言，编写处理语言文字的应用很便捷，但编写 Web 服务时就没有 PHP 使用得那么多了。

在不同语言中，既有便于个人独立实现复杂算法的语言，也有便于多个人协作实现大型作业的语言，还有便于书写一次性使用的测试类语言。

程序设计语言的选用因使用者目的不同而不同。不同语言致力于达成不同的目的。如果把为实现高速执行而设计的 C++ 语言和为了便于代码阅读而设计的 Python 语言放到一起比较，说 C++ 语言的可读性差或者 Python 的执行速度慢，这样的争论意义并不是很大。

2.3 小结

如前所述，程序设计语言是为了给人们带来便捷。但是何为便捷，语言不同，便捷的含义也各不相同。

语言只是工具。某种语言是否适合自己，要看使用这种语言能帮助自己发挥多大的能力，而不是看这种语言是否流行，别人使用它发挥了多大能力。再进一步讲，要看通过使用它自己能做出多大成果。大家不要为他人的言语所惑，应当根据自己的实际情况选择好的工具。

第3章

语法的诞生

程序设计语言有许多的规则。

为什么有这么多的规则呢?

本章通过比较规则较少的语言，来揭示规则是如何产生的。

3.1 什么是语法

程序设计语言中有各种各样的规则。比如，乘法运算比加法运算优先级高，所以 1+2*3 这样书写的算式是先计算 2*3 的。语法就是程序语言设计者规定的解释程序编写方式的一系列规则。在第 2 章中，我们讲到程序设计语言是为了带来便捷而创造的，那么语法又会是为何而创造出来的呢[①]？

本章我们来讨论乘法运算和加法运算的规则。当今主流的算式表达方法很复杂，我们来讲解一下简单一些的 FORTH 语言和 LISP 语言。FORTH 语言基本上是没有语法的，而 LISP 语言通过括号来表现代码的结构。FORTH 语言和 LISP 语言所具有的功能是当今程序设计语言重要的组成部分。

运算符的优先顺序

大家使用的语言，肯定可以表达像 1+2*3 这样的加法和乘法运算。运算符指的就是加法运算里的 + 号和乘法运算里的 * 号。

然而，1+2*3 这样的源代码，先是 1 加 2 再把结果与 3 相乘即

① 语法和句法有什么差别呢？也许有人要问这个问题。两者都是编写程序时要遵循的规则，只是句法的含义仅限于较小的范围。比如，称 if 语句为句法是很自然的，但称运算符的优先顺序为句法就很不自然了。第 4 章还会讨论诸如 if 语句、for 语句这样的控制句法，所以本章只使用语法一词。

(1+2)*3 呢，还是先 2 乘 3 再把结果与 1 相加即 1+(2*3) 呢？这个顺序是怎么规定的呢？

答案是怎么方便就怎么规定。比如，以前有些计算器计算 1+2*3 的结果有可能是 9，也就是执行的 (1+2)*3。而现在大家使用的程序设计语言大多数的计算结果都应该是 7。这是因为程序语言设计者制定了乘法优先级高于加法先行计算这一规则。考虑到这和四则运算的法则是一致的，大家理解起来一点也不费劲。

那么，对于除法运算 9/3/3，是 9/(3/3) 呢，还是 (9/3)/3？这又是如何规定的呢？大部分的语言中都是按照 (9/3)/3 来计算 。这是因为程序语言设计者制定了这样的规则：在左结合运算符除号出现并列的情况下，运算从左开始计算。

语法是语言设计者制定的规则

语言设计者制定的规则就是语法。语法因语言而异。运算符的存在类型也因语言而异。比如 C 语言中的赋值符 = 是运算符。但它与普通的加减乘除运算符不同，是右结合运算符。所以源代码中 X = Y = 1 这样的语句，被解释为 X=(Y=1)[①]。

3.2
栈机器和 FORTH 语言

FORTH 语言开发于 1958 年左右[②]，是一种几乎没有语法的语言。

其设计者 Charles H. Moore 说 FORTH 是最简单的计算机语言[③]。他

---

① 与这不同的是，Python 语言中的赋值符 = 不是运算符而是一种句式，在运算式中不能出现。

② 于 1969 年发布。

③《言語設計者たちが考えること》, Federico Biancuzzi、Shane Warden 编／伊藤真浩、顷末和义、佐藤嘉一、铃木幸敏、村上雅章译，O'Reilly Japan，2010 年，p.66。中文版为《编程之魂：与 27 位编程语言创始人对话》，电子工业出版社于 2010 年 4 月出版，闫怀志译。

认为，世上所有的程序设计语言都具备一定的可读性，但初次接触某种语言的人常常感到困惑，这是由于它们的语法往往晦涩难懂且变化多端。而 FORTH 语言将语法控制到最少的程度缓和了这一问题[①]。

我们试着讲一下 FORTH 这门语言，看一下它和现在的语言有什么不同。

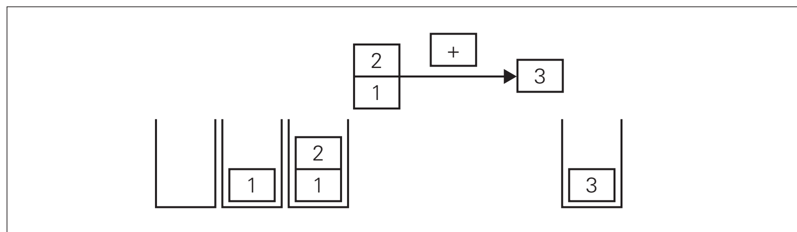
计算的流程

那么号称语法最少的 FORTH 语言是如何编写代码的呢？ FORTH 语言中，1 加 2 是这样书写的：

```
1 2 +
```

我们来看一下这行代码是如何执行的。作为其一大特点，FORTH 语言使用了被称为栈的数值预存空间，如图 3.1 上方开口的格子所示。代码从头开始执行，处理器首先遇到的是数值 1, 于是在栈中放 1。然后遇到数值 2，再次把 2 存进栈。最后遇到 + 号。这个符号被定义的功能是，从栈中取出前面两个数值，再把其相加的结果存入栈。于是，从栈中把 1 和 2 取出来，加法运算后的结果 3 被放入栈中。

图 3.1 利用栈的运算逻辑



如何表达计算顺序

这种语言是如何区别 1 加 2 再与 3 相乘即 (1+2)*3 和 2 乘以 3 再与 1 相加即 1+(2*3) 这两种计算的呢？

① 《言語設計者たちが考えること》，pp. 70-71。

1 加 2 再乘以 3 可以表达为：[1]

```
1 2 + 3 *
```

2 乘 3 再加 1 可以表达为：

```
2 3 * 1 +
```

基本上与日语中的语序是一致的 [2]。如果 1、2、3 的顺序很重要的话，也可以进行如下表达：

```
1 2 3 * +
```

即 1 和 2 乘以 3 的结果相加。在 FORTH 语言中不需要括号也不需要导入优先次序的规则就可以表达计算顺序。

现在仍然使用的栈机器

如今，估计大家很少再用 FORTH 这种基于栈的语言直接编写代码了。但是在一些不常见的场合仍然会看到它的身影。比如，Java、Python、Ruby 1.9 这些语言使用了栈机器型的 VM。VM 执行的命令行和 FORTH 语言是一样的。用 Python、Ruby 或 Java 等语言写出来的程序，在机器内部先被转换（编译）成像 FORTH 语言一样的程序，然后再运行。

我们用 Python 语言实际执行看一下。通过 Python 语言自带的库文件 dis，我们可以显示 VM 执行的命令行 [3]。

```
Python
>>> import dis
>>> dis.dis(lambda x, y, z: (x + y) * z)
 0 LOAD_FAST                0 (x)        ❶
```

① 希望实际执行一下 FORTH 语言的读者可以到笔者制作的网页上尝试一下：http://nhiro.org/learn_language/FORTH-on-browser.html

② 日语中以上两个算式的表达分别是：1 と 2 を足した（+）ものに 3 を掛ける（*）、2 と 3 を掛けた（*）ものに 1 を足す（+）。相关数值和运算符在日语语句中出现的次序与表达式是一致的。——译者注

③ 行末编号是笔者加上去的。

```
 3 LOAD_FAST              1 (y)      ❷
 6 BINARY_ADD                        ❸
 7 LOAD_FAST              2 (z)      ❹
10 BINARY_MULTIPLY                   ❺
11 RETURN_VALUE
```

(X+Y)*X 这行代码，被依次编译为命令行：❶将 X 压入栈→❷将 Y 压入栈→❸把栈上两数相加→❹将 X 压入栈→❺把栈上两数相乘。这和 FORTH 一样，是按 X Y + Z * 的顺序排列的。

3.3 语法树和 LISP 语言

前文讲到，FORTH 语言不需要使用括号或者优先次序就可以表达计算顺序。现实中，有的语言总是需要用括号标示完整的意思单元，比如 1958 年诞生的 LISP 语言。

计算流

LISP 中 1 加 2 用代码表达如下：①

```
(+ 1 2)
```

首先是一个括号，接着是加号命令，用空格分隔后，跟上进行相加运算的对象。

如何表达计算顺序

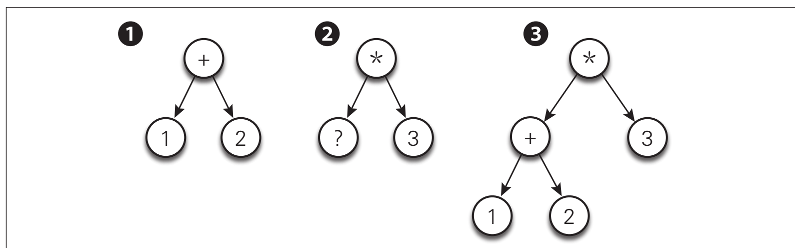
我们来看如何用代码表达 1 与 2 相加的结果与 3 相乘。把［相乘、（什么）、3］中的（什么）换成（相加、1、2），我们得到：

① 希望实际执行一下 LISP 语言的读者可以到笔者制作的网页上尝试一下：http://nhiro.org/learn_language/LISP-on-browser.html。

```
(* (+ 1 2) 3)
```

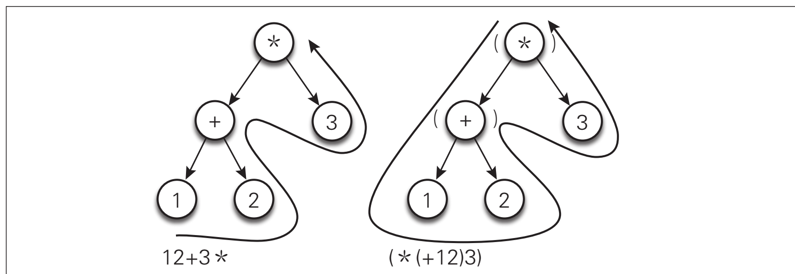
这个过程如图 3.2 所示。❶为（1 与 2 相加），在 LISP 语言中，代码表示为（+ 1 2）。图 3.2 把命令放在上方，把（什么）这部分放在下方。❷为［（什么）与 3 相乘］。（什么）这部分用？表示。❸为把（1 与 2 相加）代入（什么）这部分的结果。图 3.2 这样的结构称为语法树。

图 3.2　语法树的结构



我们把这个语法树和 FORTH 语言和 LISP 语言的代码放在一起比较一下。

图 3.3　FORTH 和 LISP 的语法树是相同的



可以看出，对于这两种语法简单的语言，它们只是在这个语法树上按不同的规则遍历而已。两者的代码看起来差别很大，但实际上所用的树结构是相同的。

现在仍然使用的语法树

当今的语言仍然在使用语法树。我们用 Python 语言实际执行一下。通过 Python 语言自带的库文件 ast，我们可以查看特定的代码被转换成

怎样的语法树[①]。

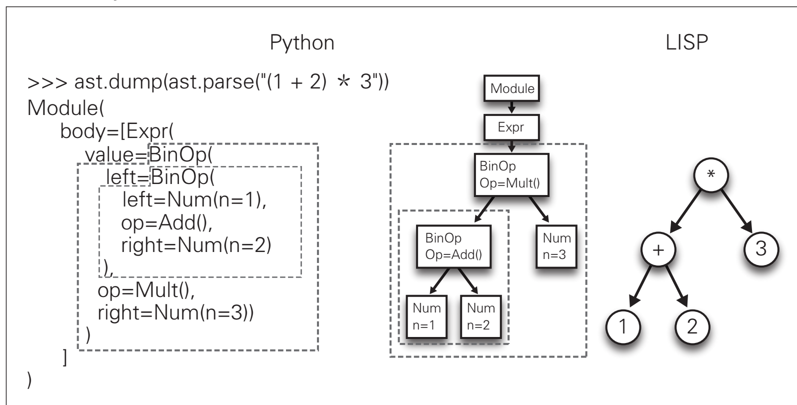
```Python
>>> import ast
>>> ast.dump(ast.parse("1 + 2"))
Module(
  body=[Expr(
    value=BinOp(
      left=Num(n=1),
      op=Add(),
      right=Num(n=2))
    )
  ]
)

>>> ast.dump(ast.parse("(1 + 2) * 3"))
Module(
  body=[Expr(
    value=BinOp(
      left=BinOp(
        left=Num(n=1),
        op=Add(),
        right=Num(n=2)
      ),
      op=Mult(),
      right=Num(n=3))
    )
  ]
)
```

我们将其和 LISP 比较一下。Python 语言这边看起来更为错综复杂，但我们还是能察觉到两者有着共通的结构。图 3.4 中 BinOp op =Mult( ) 表示乘法运算，与 LISP 的 * 相对应；BinOp op =Add( ) 表示加法运算，与 LISP 的 + 相对应。Num n=1 即为数值 1，这与 LISP 的 1 相对应。

① ast 是 abstract syntax tree 的简写，意为抽象语法树。在语法复杂的语言中，语法树是包含很多细节的语法结构表达形式。把这种形式以更简洁的形式表达出来就是抽象语法树。

图 3.4 Python 语言与 LISP 语言的语法树比较



专栏

要确认理解是否正确，首先得表达出来

假设你正在学习一些知识，并自我感觉已经理解了。那么，你到底是真正理解了呢，还是感觉自己已经理解了呢？这个仅凭自己苦思冥想是不行的。为了验证理解正确与否，需要表达出来。只能基于自己的理解说出自己的观点，然后让第三方来判断和检验。比如学习英语，就要在别人面前使用自己学到的英语，同时观察别人的反应。不这样做的话，就无法知道自己是否真正掌握了英语。

程序员一直受益于这一点。如果是写文章，写出来的东西即使有错误也可能没人指出来，或者根本没有人看你的东西。但是写程序不一样，语言处理器会事无巨细地做错误检查并指出。这和与人打交道不同，只要你方便，它总是有足够的时间和耐心陪你一起。

一旦出现程序错误，很多人可能会惊慌失措。其实那只是语言处理器在仔细阅读了你的程序后，告诉你它哪里不明白而已。只有理解了这一点，才能和语言处理器打交道。

LISP 语言语法简单，代码与语法树容易理解并且对应比较直观。此外，它还具有宏这样的语法树替换机制。这两个特点催生了程序设计语言进化路上发生的结构化编程等一系列现象。详细内容我们将在第 4 章讲解。

3.4 中缀表达式

在 FORTH 语言中，我们用 1 2 + 来表达 1 加 2，运算符放在运算对象的后面。在 LISP 语言中，1 加 2 表达为 (+ 1 2)，运算符放在运算对象的前面。而数学表达式中用 1+2 来表达，运算符放在运算对象的中间。像这样把运算符放在运算对象之后、之前和之中的表示法分别称为后缀表达式、前缀表达式、中缀表达式[①]。

这三种表达式只是表达方法上的约定事项而已。在程序设计语言出现以前，人们仅习惯于用中缀表达式来书写数学表达式。

FORTRAN 语言就是旨在用已经习惯了的方式编写程序。这就是为什么 FORTRAN 语言的全称为 Formula Translating System（表达式翻译系统）。FORTRAN 语言导入了运算符优先级和结合性等复杂的语法，但程序员可以用习惯的方式来编写数学表达式。

语法分析器

语法分析器是把源代码作为字符串读入、解析，并建立语法树的程序。FORTRAN 语言在编译程序时，语法分析器会将源代码的字符串转换为语法树。

语法的设计和语法分析器的实现是决定语言外在表现的重要因素。语言设计者在设计语法时，会考虑使哪些部分变得编写简便，哪些变得不容易出错，哪些能给用户带来价值等问题。

但是，从这些角度出发设计的语法在语法分析器中能否实现却是另一个问题。语法的解析方法不同，实现清晰明确的解析的规则也不同。

---

① 前缀表达式和后缀表达式也被称为波兰表示法和逆波兰表示法，得名于最先研究它们的波兰人 Jan Lukasiewicz。前缀表达式中的括号并不是必需的，因为只要知道 + 和 * 取的是两个运算对象，即使省略了括号，表达式 * + 1 2 3 的含义也是可以理解的。LISP 语言中，有三个以上运算对象的情况也可以写成 (+ 1 2 3 4)。因为有了括号不会导致误解，省略括号是不行的。

此外，在现有的语言中引入新的功能时，有可能因为和既有的规则发生冲突而产生问题。

规则的竞争

在 C++ 语言中增加模块功能时，引入了 vector<int> 这样用不等号括起来的表达方法。这种方法在二重表达时，一侧的符号 >> 就会被解析为既有的移位运算符 >>。语法分析器要解决这个问题并不容易，所以程序员通过增加空白符的方法来避免其变成移位运算符。

专栏

当你不知道该学习什么时

应该学习什么？我们经常听到这样的问题。在回答之前，笔者想先问这样一个问题：你学习的目的是什么？这个如果不明确，提建议就无从下手。

我们生活在一个信息爆炸的时代。不管三七二十一统统都学，这样的学习策略已经不再适用。必然要有所学，有所不学。这时，我们就要事先明确自己到底想做什么，然后再去学习能够达成这一目标的知识。

不知道自己要做什么？或许你从一开始就在苦思，想做一件完美的事情。一件从没有人想到过的，能获得别人赞誉的事情。如果最初的设想太过宏大导致无从下手，那么这一设想就永远不可能实现。还不如从小事做起，从简单的事情做起。在这个过程中，你可以逐渐明白自己哪些已经能做、哪些还不能做，如果要做还要学习哪些知识。长此以往，就能培养出完成更复杂的任务的能力。

C++
```
// OK
vector<vector<int> > x;
// NG
vector<vector<int>> y;
```

当然，理想的情况是把语法分析器改良到能同时照顾好两方又不出错的程度。然而现实与理想总是有差距的。由既有规则的羁绊导致不自然的规则产生，这种现象不仅限于 C++ 语言，在其他语言中同样存在。

3.5 小结

同样是处理 1 加 2 乘以 3 这样的运算，不同语言的表达方式大相径庭。但是基本上都是用语法树来表达。语言之间的这种差异就是语法的差别，它决定了怎样的代码对应怎样的语法树。

FORTH 语言和 LISP 语言尽量精简规则。但是市场追求的不是规则数量多么少、多么简单。相比之下，FORTRAN 语言大量导入了诸如乘法运算符优先级高于加法这样的决定性规则，重视编写的便利性。这种设计理念大获成功，与 LISP 语言和 FORTH 语言相比，FORTRAN 语言的风格为更多的人所接受。

大家在编写程序时，是不是时常抱怨为什么有这么别扭复杂的编写规则？现代的大多数语言都崇尚 FORTRAN 语言风格，追求简单便利的编写规则。然而，设计不存在任何解析矛盾的语法体系是十分困难的。随后要再融入新的语法时不与既有的语法发生冲突，这个尤其困难。正因为如此，现实中程序设计语言仍然保留有不少别扭复杂的编写规则。

第4章

程序的流程控制

程序设计语言中有 if、while、for 等用来控制程序流程的语句。
为什么会有这些控制语句呢?
本章我们将通过比较没有控制语句的汇编语言和带有控制语句的 C 语言，来探讨控制语句是如何产生的。

4.1 结构化程序设计的诞生

从第 3 章我们了解到，为了能使用更加自然的表达方式来书写算式，程序设计中引入了乘法运算优先级高于加法运算这样的规则（语法）。

20 世纪 60 年代后期，在提倡规则让读写程序更轻松的时代潮流中，结构化程序设计应运而生。时至今日，大家对 if、while 这样的语句早已习已为常。结构化程序设计的初衷正是通过导入这些语句使代码结构的理解变得简单。

虽说这些语句的导入是为了使代码结构更简单，但现在大家都觉得这是理所当然的事，也许一下子也觉察不到它带来的变化了。那么，我们拿它和不使用 if 或 while 语句的代码比较一下吧。

4.2 if 语句诞生以前

如果没有 if 语句该如何编写程序呢？我们首先来考察一下这一问题。

为什么会有 if 语句

本章我们使用一种非常原始的程序设计语言——汇编语言。汇编语言中是没有 if 语句的，但是从 C 语言很容易就能编译成汇编语言。接下

来，我们用 C 语言先编写带 if 语句的代码，再试着将其编译成汇编语言看一下[①]。

C 语言下的源代码如下所示，其含义是如果 X 等于 456 则做相应处理[②]。

C语言
```
int main(){
  int x = 123;
  /* if语句前 */
  if(x == 456){
    /* if语句中 */
  }
  /* if语句后 */
}
```

编译后输出如下汇编语言代码。

汇编语言
```
_main:
        ......
        movl    $123, -8(%rbp)      ❶
        # if语句前                  ┐
        movl    -8(%rbp), %eax      ┘❷
        cmpl    $456, %eax
        jne     LBB1_2              ❸
        # if语句中                  ❹
LBB1_2:                             ❺
        # if语句后
        ......
```

我们试着来解读一下：首先，把 -8(%rbp) 理解为原来代码中的 x。❶句把数值 123 代入 x，❷句将 x 的值移存到临时场所后，把它和数值 456 相比较。

① 通过编译转换成汇编语言，然后汇编到机器语言，最后链接成一个可执行文件，现在一般把这一整串的动作统称为编译。这里说的编译仅指转换成汇编语言这一个步骤。

② 实际实验的代码里，通过使用 __asm__ 在汇编语言里嵌入了注释。这里为了简洁易读做了改写。请参照本书文前的“本书构成”部分获取可执行的源代码。

接下来的❸句是关键，它表示在前一句的比较中，如果两边不相等则跳转至 LBB1_2 处。换句话说，如果两边相等则不跳转接着执行下面的命令。❹句就是 if 语句中的代码，它只在两边相等时被执行。不相等时程序跳转至 LBB1_2（即❺句处），❹句不被执行[①]。

这种满足条件后跳转的命令很早就有。比如 1949 年发明的 EDSAC 就有“特定内存值大于零时跳转”和“特定内存值为负时跳转”这两条命令[②]。

为什么会有 if...else 语句

大家在学习 if 语句时，想必同时也把 else 和 else if 语句配套地学习了吧[③]。没有 else 和 else if 程序就没法写了吗？非也！本节我们来看一下同样没有 else 和 else if 语句的汇编语言。

汇编语言中的表达方式

我们先在 C 语言中实现 else 语句，然后把它编译成汇编语言。这是一段处理 x 值为正为负或为零时的代码。

C语言
```
/* if语句前 */
if(x > 0){
  /* 为正时的处理 */
}else if(x < 0){
  /* 为负时的处理 */
}else{
  /* 为零时的处理 */
}
/* if语句后 */
```

与前面一样，这段代码编译后结果如下。

---

① 这几个命令的意思分别是：movel=move long integer，cmpl=compare long integer，jne=jump if not equal。

② 准确来讲，不是内存而是累加器（accumulator）。本书把内存一词理解为记忆装置并用在全书中。

③ C 语言中，else if 不是独立的语句，而是 else 后紧跟着的 if 语句。这个表达在各种语言中不尽相同，Perl 语言和 Ruby 语言中有 elsif，Sh 语言和 Python 语言中有 elif 这样专门的关键字。

汇编语言

```
_main:
        ……
        # if语句前
        movl    -8(%rbp), %eax
        cmpl    $0, %eax              ┐
        jle     LBB1_2                ├─❶
        # 为正时的处理   ❷             ┘
        jmp     LBB1_5 ❸
LBB1_2:                ❹
        movl    -8(%rbp), %eax        ┐
        cmpl    $0, %eax              │
        jge     LBB1_4                ├─❺
        # 为负时的处理   ❻             ┘
        jmp     LBB1_5 ❼
LBB1_4:                ❽
        # 为零时的处理   ❾             ┐
LBB1_5:                ⓫              ├─❿
        # if语句后                     ┘
```

我们按顺序来读一下：❶句指如果 x 小于或等于 0 时跳转至 LBB1_2（❹句），❷句是为正的处理，❸句跳转至 LBB1_5（⓫句）。接着，❺句指 x 大于或等于 0 时跳转至 LBB1_4（❽句），❻句是为负的处理，❼句跳转至 LBB1_5（⓫句）。最后，❿句是为零的处理。[①]

那么，就实际中 x 为正，x 为负，x 为零的情况，我们来追踪下程序是如何执行的。

为正时，❶句的“小于等于零则跳转”不成立，因此不跳转而继续执行❷句中为正时的处理，然后执行❸句中跳转至⓫句，程序结束。

为负时，❶句的“小于等于零则跳转”成立，跳转至❹句，❺句的大于等于零跳转不成立，故不跳转而继续执行❻句中为负时的处理，然后执行❼句中的跳转至⓫句，程序结束。

为零时，❶句的“小于等于零则跳转”成立，跳转至❹句，❺句的

① jle 是 jump if less or equal 的略称，实现小于等于零时的跳转。Jge 与之相反，是 jump if greater or equal 的略称。jmp 是 jump 的略称，表示无条件跳转。

“大于等于零跳转”成立，故跳转至❽句，在❾句执行为零时的处理。

由上可见，本来要表达如果等于某值则执行某事的逻辑的，却不得不表达为如果不等于某值则跳转至某处执行某事。如此这般条件颠倒，看起来实在是有些混乱[①]。

C 语言中的表达方式

这种不使用 else 语句的书写方式在 C 语言中不可能实现吗？不是的。C 语言中，如果使用跳转至指定行的命令 goto 语句的话，这个一样可以实现。实验中的实现代码如下所示。goto END; 语句意指跳转至标示有 END: 的一行。

C 语言
```
void not_use_if(int x){
  if(x <= 0) goto NOT_POSITIVE;
  printf("正数\n");
  goto END;
 NOT_POSITIVE:
  if(x >= 0) goto NOT_NEGATIVE;
  printf("负数\n");
  goto END;
 NOT_NEGATIVE:
  printf("零\n");
 END:
  return;
}
```

使用 if...else 语句的好处

众所周知，C 语言程序设计中 else 语句的使用不是必不可少的，替代方案是使用 goto 语句。其功能是跳转至指定的某行，这很好理解。

那么，上面的代码便于理解吗？至少在笔者看来，这段代码烦杂、结构差，如果不瞪大眼睛仔细读很难理解。我们将其和直接使用 if...else 语句的代码比较下，来看看哪种方式更便于理解。

C语言
```
void use_if(int x){
```

① 也有条件不颠倒的写法，由于篇幅所限，在此不介绍它的代码实现了。

```
  if(x > 0){
    printf("正数\n");
  }else if(x < 0){
    printf("负数\n");
  }else{
    printf("零\n");
  }
}
```

导入 if...else 语句的好处正在于此。针对条件为真为假的不同情况分流处理，这样的模式在程序设计中屡见不鲜。为了能简洁地表达并方便轻松地阅读这种逻辑，于是引入了 if...else 语句这种新的规则。

程序设计中 else 语句并不是必需的。但是，笔者还是乐于使用 else 语句，毕竟这能让程序编写更加轻松。另外，笔者也希望别人能使用它，毕竟这也能让理解程序更轻松。

4.3 while 语句——让反复执行的 if 语句更简洁

下面我们来考察一下 while 语句。while 语句是指满足条件时反复执行某区间中的代码[①]。

使用 while 语句的表达方式

首先，我们来看一段使用了 while 语句的代码，它表示只要满足条件 x>0，就会反复执行打印显示 x 并减 1 的操作。

C语言

```
void use_while(int x){
  printf("use_while\n");
  while(x > 0){
    printf("%d\n", x);
```

① 严格来讲，这只是在 while 语句后存在多行代码的情况。有时候 while 语句后面也可能只有单行代码。

```
    x--;
  }
}
```

不使用 while 语句的表达方式

要达到同样的目的，不使用 while 语句可以实现吗？答案是肯定的。赶紧来看看下面的代码吧，它表示的是条件不满足时跳转至 END_LOOP，然后打印显示 x 并减 1，再跳转回条件判断语句前。

```c
void not_use_while(int x){
  printf("not_use_while\n");
 START_LOOP:
  if(!(x > 0)) goto END_LOOP;
  printf("%d\n", x);
  x--;
  goto START_LOOP;
 END_LOOP:
  return;
}
```



很多语言定义了用于中断循环的 break 语句，执行 break 语句后立刻从循环中跳出。这个动作和 goto END_LOOP 是一样的。

像这样，while 语句和 break 语句做的只是那些只要有 goto 语句就能做的事情。while 语句带来的附加值不是新的功能，而是程序的易读性和易写性。

goto 语句是很强大也很容易理解的概念，但是过于原始。如果随意使用 goto 语句，程序将彻底散了架。再好的马，不配上缰绳也不能为人们所用，goto 语句的使用也需要加以限制，这样才便于代码的理解。if...else、while、break，这些就是加以限制了的 goto 语句①。

① 参见艾兹格·迪科斯彻“go to statement considered harmful”, *communications of the ACM*, Vol.11, No.3, ACM, 1968, p.3。

4.4 for 语句——让数值渐增的 while 语句更简洁

笔者曾有耳闻，大学里初学 C 语言时，有些人提出，有了 while 语句，for 语句不要也可以。有这种质疑其实并不奇怪，因为实际上 for 语句能实现的功能用 while 语句已经能够实现了。

使用 for 语句的表达方式

我们来考察下面的 for 语句，它表示 i 在 0 至 N 的范围内按 1 递增同时打印显示。

C语言
```
for(i = 0; i < N; i++){
    printf("%d\n", i);
}
```

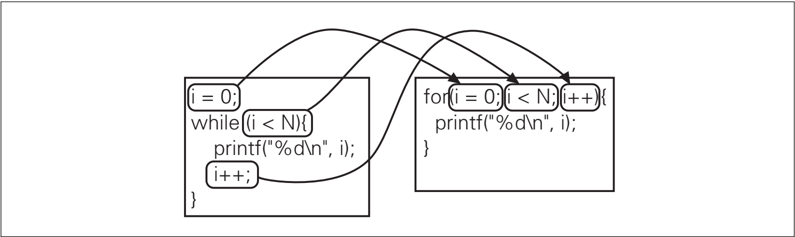
不使用 for 语句的表达方式

同样的逻辑使用 while 语句来表达，就变成了下面这样。

C语言
```
i = 0;
while(i < N){
    printf("%d\n", i);
    i++;
}
```

对在 0 至 N 范围的某数做某种操作，这样的需求时常能碰到。比如，要对数组 xs 中全部数值做某种处理，即，要对从 xs[0] 到 xs[N-1] 范围内的各个元素做处理。如果用 while 语句来表达，就需要在循环体外写 i=0，循环条件写 i<N，循环体最后写 i++。代码分散在三处，对于阅读代码的人来说，原本的意图没那么直观。用 for 语句就不同了，相关代码更加紧凑，代码阅读者很容易就能理解循环的意图（图 4.1）。

图 4.1　三处散落的代码在 for 语句中只存在于一处



像这样，带有初值、递增值和终值这三组数的 for 语句，早在于 1958 年发明的 ALGOL 58 语言里就已经出现了[①]。

ALGOL 58
```
for I : = 0 ( 1 ) N; ……
```

foreach——根据处理的对象来控制循环操作

for 语句已经有了新的发展，这就是目前许多语言里采用的 foreach 句型。在 Java 语言里被称为扩展 for 语句，而在 Perl[②]、PHP、C# 等众多语言里被叫做 foreach 语句。本书为了体现其与 for 语句的区别，把它称为 foreach 语句[③]。

while 语句通过条件判断来控制循环操作，for 语句通过循环次数来控制循环操作[④]，而 foreach 句型则是通过处理的对象来控制循环操作。

在没有 foreach 语句的 C 语言里，for 语句常常被用来实现对数组里各元素的处理操作。foreach 的句型的产生，就是为了方便编写对某对象内所有元素进行某种处理的代码。

① 参见 A. J. PERLIS and K. SAMELSON, “Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming”, *Numberische Mathematik*, Bd.1, S.41-60, 1959, p.50.

② 在 Perl 语言中，for 和 foreach 为同义词，为了方便阅读，两者在语言中都有提供。

③ Python 语言的 for 语句就是 foreach，它反而没有相当于C语言里功能的 for 语句。

④ 准确来讲，for 语句也是通过条件判断来控制的，但它主要体现的还是对循环次数的跟踪。

```Java
// 数组
int[] items = new int[]{1, 2, 3, 4, 5};

// 用一般的for语句输出各元素
for(int i = 0; i < items.length; i++){
    int item = items[i];
    System.out.println(item);
}

// 用扩展的for语句输出各元素
for(int item: items){
    System.out.println(item);
}
```

如上，for 语句表达的是在 0 至数组 items 长度范围以内，对 i 按 1 递增同时打印显示数组 items 第 i 个元素。使用了 foreach 语句后，意思就变成了将数组 items 里的各个元素都打印显示出来，相当地简洁易懂。

4.5 小结

本章我们学习了 if 语句、while 语句、for 语句等用来控制程序流程的语法规则。虽然不使用这些语句也可以编写程序，但是使用它们会让我们的程序变得更容易理解。所以，为了写出简洁易懂的程序，请大家多使用这些程序流程控制语句吧。

第5章 函数

为什么会有函数呢？
本章我们将学习函数诞生的过程。
同时，我们会学习因函数的发明而出现的递归调用，了解它是怎么样一种程序行为，以及它在何时使用。

5.1 函数的作用

在第 4 章中，我们学习了 if 语句、for 语句、while 语句等产生的原因。本章我们来学习函数，即把代码的一部分视作有机整体，然后切分出来并为之命名的程序设计机制[①]。

函数为什么必不可少呢？有没有因为没有函数而不能编写的程序呢？答案是否定的。尽管没有函数也可以编写程序，但使用函数编写程序将变得更轻松简便：因为它便于理解和重复使用。

便于理解——如同一个组织

把代码切分为多个函数，如同将一个大的组织按部门划分开。在一个小的程序中，函数的优越性体现不出来，这和没必要把几个关系融洽的人组成一个部门让大家互相熟悉是一个道理。人数很少的话，圈子里的每个人都熟知其他人的姓名、长相和特长。同样，代码数很少的情况下，很容易就能把握每一行执行什么功能。

问题是，人数一旦多起来怎么办。这样一来，把握方方面面的事情变得越来越困难。因此，可以把某些人视作一个单独的小组并为这个小

① 这种机制在不同时期和不同语言中，有事务、程序（procedure）、子程序（subroutine）等不同的叫法。但大多数人都习惯称它为“函数”。另外，类似的机制还有方法（method）。关于这个我们会在第 11 章讲解，这里我们为简单起见，将函数和方法等同视之。

组取个名字，比如财务部、某开发组等。

程序设计上也一样。源代码的行数多起来后要把握整体就变得困难起来。因此，把一定行数的代码视作一个整体并为之取名，这就是函数。

便于再利用——如同零部件

构建函数类似于将小的零部件组合起来制造大的零部件。比如，无线遥控玩具车里有碱性电池和电机，把碱性电池和电机放到指定的位置，就能把玩具车组装起来。

事实上，电机中还包含线圈、磁铁和整流子[①]，而碱性电池中有二氧化锰、锌和氢氧化钾。

要无线遥控玩具车，当然首先要制作电机和电池。为此，就要理解电机是如何将电流转化成旋转运动以及电池中发生了怎样的化学反应从而产生了电。现实中，把二氧化锰、锌和氢氧化钾配置包装起来并销售的东西已经存在，那就是碱性电池，在便利店等地方很容易就能买到。电机也能在塑料模型用品店里轻松买到。

函数也一样。将数十行、数百行代码作为函数，通过简单地调用就能使用它的功能[②]。

另外，电池这一命名方式有助于理解使用了电池的系统功能。即使不知道电池内部的详细结构与工作原理，只要知道“电池就是能产生电的东西”这一点，小学生也能组装无线遥控玩具车。并且，玩具车一旦行驶缓慢就会让人想到是不是电池快没电了。这个道理和函数有助于理解程序是类似的。

程序中再利用的特征

编写程序和制造物理实体有一个很大的区别，那就是重复使用零部件时所产生的成本的类型。

---

① 这是电机中非常重要的零部件之一，它通过切换线圈中电流的方向保证线圈持续地朝一个方向旋转。

② 也许有人说，不理解其中内容就拿来使用不太好。这种看法是可以理解的，但与本文论述无关，这里不作讨论。

看个例子，假如要为整栋公寓楼所有房间的水龙头安装净水器，公寓楼有 100 个房间的话就需要配备 100 个净水器，相应地，有 200 个房间的话就要 200 个净水器。房间数量越多，所耗费的资金和空间资源也就越多。

我们再回到程序中，假如要用某个函数来处理列表中所有的数据，有 100 个数据的话就要调用此函数 100 次，有 200 个的话就要调用 200 次。数据量越多，所需的执行时间也就越长。但是，函数的实现只需一个就足够了，调用 200 次并不需要将此函数实现 200 次。通过把需要反复执行的操作封装成函数，进而多次调用，可以确保源代码紧凑且清晰。

把相同的操作封装在一起的好处不仅仅在于使程序更简短，也在于能使阅读程序的人无需反复读取相同内容的源代码。从冗长的程序中切分出反复使用的代码将其封装成一个整体，程序就更容易理解了。

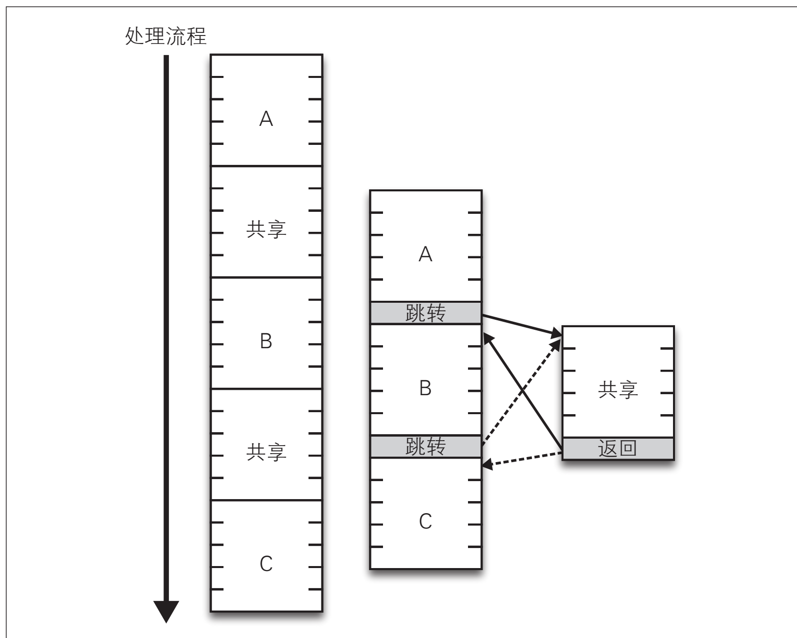
5.2 返回命令

从第 4 章我们了解到，if 语句、while 语句、for 语句全部都可以借助 goto 语句实现。但是从源代码再利用的角度来看，仅仅依靠 goto 语句是不够的。

goto 语句无法将程序返回原来的位置。我们期望的运行是，执行跳转语句时记住这一位置，之后碰到返回语句时又能跳转回到该位置后面的语句。

有了返回原来的位置这样的命令，代码的再利用成为可能。一个程序中有几处执行相同的操作时，就可以把这些操作封装在一个地方了（图 5.1）。

图 5.1 返回命令使代码再利用成为可能



函数的诞生

把反复使用的命令封装在一起再利用，这种需求在很早以前就有了。1949 年的 EDSAC 就使用了带有这一功能的技术[①]。

当时，程序的命令和数据完全都存储在内存中，修改程序就如同把数值代入变量中一样简单。通过修改程序中跳转命令的跳转目的地，就能使函数调用后返回原来的位置[②]。

- 1: 将 110 处跳转命令的跳转目的地改写为 3 处

① 当时还没有为之取名以便于理解。1949 年编写的“The Square Program”中，92 处和 114 处的指令将原先放置在 75 处的跳转命令的跳转目的地改写了。这里仅对该技术作简单说明，详细的源代码和解释可以参考该 PDF 文件链接：http://www.cl.cam.ac.uk/~mr10/edsacposter.pdf。

② 也许有人说，这个因为没有返回值，所以不是函数而是子程序（subroutine）。这么说没错，但本书将子程序也归为函数进行说明。

- 2: 调用函数（跳转至 100 处）
- 3: 下一个命令
- ……
- 51: 将 110 处跳转命令的跳转目的地改写为 53 处
- 52: 调用函数（跳转至 100 处）
- 53: 下一个命令
- ……
- 100: 函数操作
- ……
- 110: 返回（跳转至 0 处）

就这样，函数诞生了[①]。

记录跳转目的地的专用内存

在函数调用前修改返回命令的跳转目的地时，函数调用者必须同时知道跳转目的地在哪里和返回命令所在地在哪里。这是很难办到的。假如在函数中增加几行代码，返回命令的位置就会相应地往后挪一些。这样一来，就不得不修改调用这一函数的全部代码。

后来出现了稍微改良过的方法，即创建用来事先记录返回目的地的内存空间，并设计能跳转到该内存空间里记录的地址的命令。这样，即使函数调用前不知道返回命令所在地也没关系了[②]。

- 1: 将 3 写入返回目的地内存
- 2: 调用函数（跳转至 100 处）
- 3: 下一个命令
- ……
- 100: 函数操作

① 其他领域也在使用函数这个概念。比如，作为数学用语的函数（function）是莱布尼茨于 1673 年最早使用的。在理论计算机科学领域，1930 年发明的 λ 演算也使用了函数这个概念，通过建模说明计算的本质。

② 这里的返回目的地内存通常位于寄存器的高速存储装置中。

- ……
- 101: 返回至返回目的地内存所记录的地址

然而，这种方法也有一个问题。当调用函数 X 期间又调用了函数 Y 时，返回目的地内存被写覆盖，函数 X 执行之后应该返回的目的地地址就找不到了。这时该如何处理呢？

专栏

函数命名

说到函数为操作命名的好处和实现方法，其实和函数之外的其他因素也有关系。

使用函数给操作命名的做法，就是用便于理解的字符串取代数值，来表示操作开始时内存的位置。这和变量一样。变量的诞生，就是为了用字符串替代数值来表示存储了某个值的内存的位置。

关于名字和作用域的相关内容，我们将在第 7 章详细论述。

栈

栈终于登场了[①]。栈是一种存储有多个值的数据结构，实现最后被存入的值最先被读取。在第 3 章介绍 FORTH 语言时我们也有提到。

那么栈具体是怎么实现的呢？首先，决定记录栈顶位置（即最后被存入的数据的地址）的内存地址。图 5.2 中这一地址就是 42。之后每当存入新数据时将按步骤执行，42 处数值加 1 后把数据存入该数值指向的地址[②]。

最初的状态（图 5.2 ❶）如下：

- 42: 栈顶在哪（当前值：100）

① 此外还有其他方法，比如为函数分别准备记录返回位置的场所。早期的 FORTRAN 语言就采用了这种方法。但在调用函数 X 期间再次调用函数 X 时，这种方法就无法知道返回目的地了，无法实现下一节中讲的递归调用。

② 此处的 42 没有什么特殊含义。

- 100:

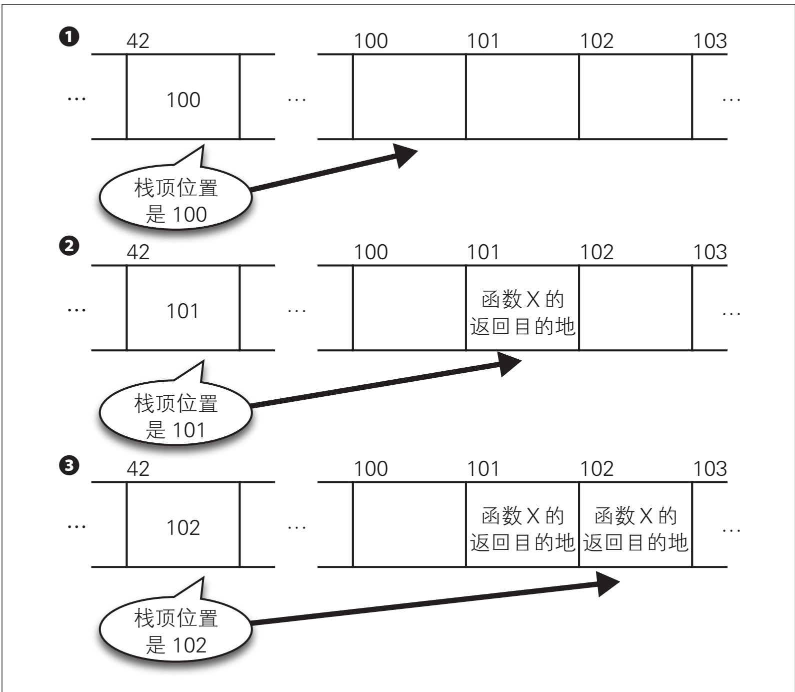
将函数 X 的返回目的地写入该栈后的状态（图 5.2 ❷）如下：

- 42: 栈顶在哪（当前值：101）
- 100:
- 101: 函数 X 的返回目的地

然后把函数 Y 的返回目的地写入栈。42 处的值加 1 后变成 102，数据被写入 102 处（图 5.2 ❸）。

- 42: 栈顶在哪（当前值：102）
- 100:
- 101: 函数 X 的返回目的地
- 102: 函数 Y 的返回目的地

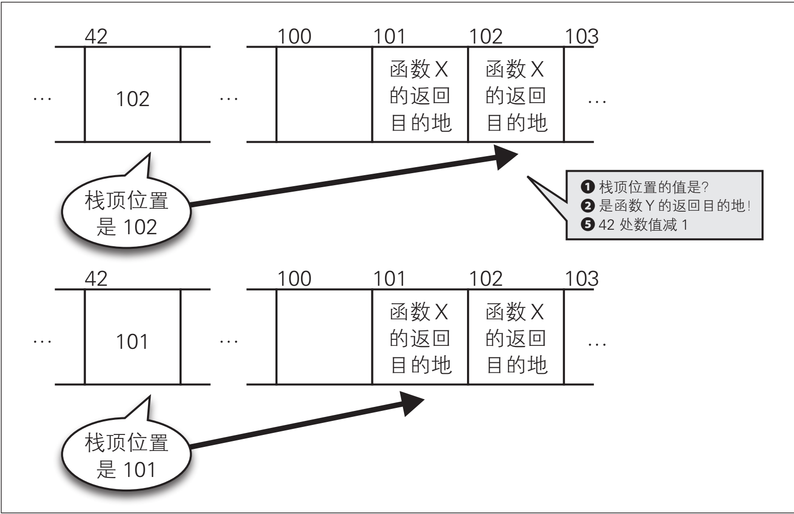
图 5.2　在栈中写入值



接下来，我们来看一下数据的读取过程。数据读取时按照步骤，先读 42 处数值指向的地址上存储的数据，再将 42 处数值减 1（图 5.3）。

- 42: 栈顶在哪（当前值：101）
- 100:
- 101: 函数 X 的返回目的地
- 102: 函数 Y 的返回目的地

▌图 5.3　从栈中读取值



这样一来，即使在调用函数 X 期间又调用了函数 Y，也不至于把函数 X 的返回目的地写覆盖，程序可以顺利地返回。

5.3

递归调用

所谓递归调用，是指函数内部再次调用当前函数的过程。过去有些语言无法实现递归调用，现在几乎所有语言都支持这一编程技术。

嵌套结构体的高效处理

递归调用是不可或缺的吗？不，当然不是。使用了递归调用的程序，也可以不用递归调用来实现[①]。

那么，递归调用这种程序设计技巧为什么会产生并一直被使用呢？这是因为，对于某些类型的操作，使用递归调用可以使程序编写变得轻松很多。这里指的是那些执行某些步骤中途又针对不同对象（参数）执行相同步骤的情况，即嵌套结构体的情况。处理嵌套的数据结构时，代码通常也会变成嵌套结构。

嵌套结构体的处理方法

在物理实体的制造中，要说某零部件是使用该零部件自己制造出来的，这是极其难以想象的一件事。初学程序设计时，很多人对递归调用感到困惑。作为处理嵌套结构的一个例子，我们来看一下为嵌套列表的全部元素求和的问题[②]。

比如，[1,2,[3,4],5] 这样一个列表，可以看作是将 [3,4] 这个列表嵌套放入 [1,2,?,5] 这个列表产生的。要为这样一个嵌套结构的列表里所有元素求和，该如何实现呢？

下面的 Python 代码使用 for 语句把列表中的元素逐个取出，如果为整数则做相加运算，执行情况如下。[③]

```
Python
def total(xs):
    result = 0
```

① 最坏情况下，自己来设计栈也是可以实现的。比如把使用了递归调用的汉诺塔求解过程用不带递归调用的方式实现。这就是一个不错的练习。

② 顺便一提，笔者认为使用阶乘计算和裴波那契数列计算这两个例子来讲解递归调用恐有不妥。这是因为，这两个计算不需要递归调用，使用普通的循环体就可以实现，并且对于裴波那契数列计算如果简单地使用递归，其性能比使用循环体要差很多。

③ 在 Python 语言中不存在 is_integer 这个函数，实际起相同作用的是 isinstance(x, int) 函数。因为这个无关正题，在这里我们选择使用一个名字更好理解的函数。

```
    for x in xs:
        # 逐个取出列表xs中的元素放进x
        if is_integer(x):
            # 如果x为整数则做加法
            result += x
        else:
            # 如果x不为整数该如何处理? ?

    return result
```

最早出来的是 1 和 2，因为它们都是整数，所以与 result 做加法，最后结果是 3。到此为止都很顺利，但接下来出现的 [3,4] 不是整数了，这个该如何处理呢?

无法用 for 语句实现

也许有人说，因为这是一个列表，在 for 语句结构中对其元素做特别加法不就行了。对于这个例子中的输入数据，这种实现方法恰巧是可行的。对于最多仅有两重嵌套的输入数据，只要用二重 for 语句就可以处理。但是，输入为三重嵌套结构的列表又会怎样呢? 此时第二个 for 语句的处理中又碰到一个列表，对这个列表该怎么处理呢?

```python
def total(xs):
    result = 0
    for x in xs:
        if is_integer(x):
            result += x
        else:
            # x为列表，所以用for语句处理
            for y in x:
                if is_integer(y):
                    result += y
                else:
                    # 再来一个列表时该如何处理呢?

    return result
```

此处即使再追加一个 for 语句，这段程序也只是针对三重嵌套结构

的处理管用，如果数据有四重、五重嵌套就无法处理了。

这种多重嵌套的数据结构并不罕见，比如 HTML 语言中的标签就有几十层嵌套。处理这样的数据结构，需要的是不管多少层嵌套都能做处理的机制，多次嵌套 for 语句是无法做到的。

使用递归调用

于是就有了递归调用。这种方法在实现对嵌套列表元素求和的函数 total 中，又调用该函数自身，如同该函数已经实现完成了一样。

```python
def total(xs):
    result = 0
    for x in xs:
        if is_integer(x):
            result += x
        else:
            # x为嵌套列表，所以用total求里面的元素的总和！
            result += total(x)

    return result
```

这样就完成了。不管对函数 total 传递几重嵌套结构体的列表，都能把它里面的元素的和求出来。

递归调用执行时的程序流

给函数 total 传递参数 [1, [2, 3], 4] 并调用会发生什么呢？我们顺着递归调用的执行过程一起来看一下。

首先，函数 total 带有参数 [1, [2, 3], 4] 被调用时，xs 为 [1, [2, 3], 4]，result 为 1（图 5.4）。

然后开始执行 for 语句循环。先把 xs 的第一个元素取出，为整数的 1，故执行 result 加 1，result 由 0 变成 1（图 5.5[①]）。

---

① 此图为简化图形，未把 x 放入其中。

图 5.4 循环开始前

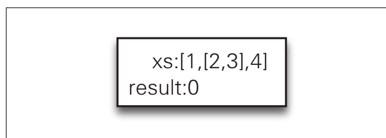
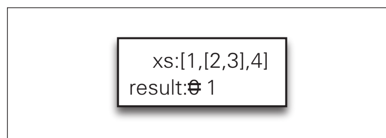
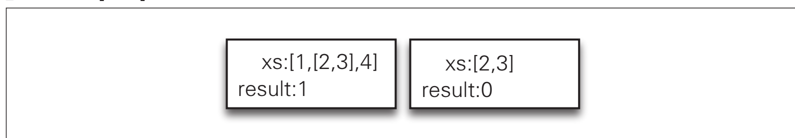


图 5.5 对 1 的操作



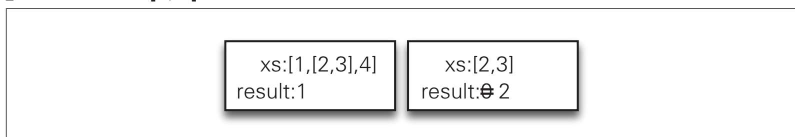
然后循环至下一个元素。把 xs 的第二个元素取出，发现是非整数的 [2, 3]。为了求解这个数组的和，把 [2, 3] 作为参数调用函数 total。在第二个调用中，xs 为 [2, 3]，result 为 0（图 5.6）。

图 5.6 [2, 3] 非整数故递归调用 total



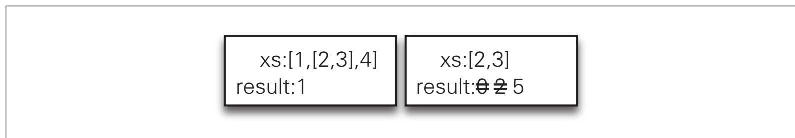
第二个调用中的 for 语句开始执行。把 xs 的第一个元素取出，为整数的 2，故执行 result 加 2，result 由 0 变成 2（图 5.7）。

图 5.7 处理 [2, 3] 中的 2



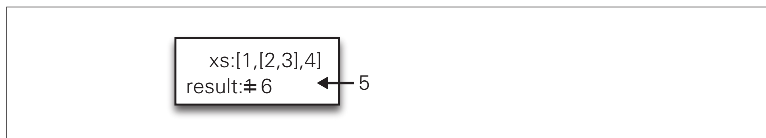
第二个调用中的 for 语句继续执行。把 xs 的第二个元素取出，为整数的 3，故执行 result 加 3，result 由 2 变成 5（图 5.8）。

图 5.8 处理 [2, 3] 中的 3



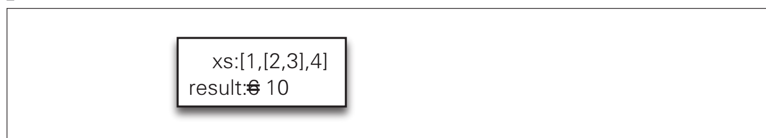
这样第二个调用中的 for 语句循环结束。循环体外有 return result，把此时 result 的值 5 返回函数调用的地方。在函数调用处，返回值与 result 相加，result 由 1 变成 6（图 5.9）。

图 5.9　返回 5 后对 1, [2, 3] 的处理结束



把 xs 的第三个元素取出，为整数的 4，故执行 result 加 4，result 由 6 变成 10（图 5.10）。

图 5.10　处理最后的 4



total 的第一次调用中的 for 语句执行完毕，result 值为 10，故将 10 返回至函数调用的地方。

这样就成功地返回了正确的结果。在这个执行过程中，针对每次函数调用都有单独的地方用来存储 xs 和 result 的值，并且在第二个 total 执行结束时第一个 total 能紧接着执行，这两点值得注意[①]。

5.4 小结

随着程序变得越来越庞大，把握全局逐渐地变得困难起来。同时，有可能需要多次用到非常相似的操作。

函数就是为解决这个问题产生的。通过在语义上把一整块代码切分出来为之命名，理解这段代码变得更加容易。此外，通过在其他地方调用这个函数，实现了代码的再利用。

伴随着函数的使用产生了递归调用这一编程技巧，它非常适合处理嵌套形式的数据。

① 在这个例子中，每次函数调用时都有各自的 result 值，这是因为 result 是函数的局部变量，在此不做深入说明。全局变量和局部变量的区别我们将在第 7 章详细论述。

第6章

错误处理

程序在执行过程中也有可能出错，出错时如何处理（错误处理）很重要。错误处理的方法大体可分为两种：使用返回值和使用异常（异常处理）。除 C 语言外，多数语言都支持异常处理。

本章我们将回顾异常处理的发展历史，分析它为什么是今天的形态，与使用返回值的方法相比它的优势所在，以及其仍有待解决的问题。

6.1 程序也会出错

当今很多语言都支持异常处理这种机制，如 Java、C++、Python、Ruby 等语言。但是 C 语言是不支持这种机制的。因此，以 C 语言入门学习程序设计的人不少都对为何需要异常处理这个问题认识模糊。本章会说明异常处理产生的目的、它是如何发展的，以及异常处理中仍然尚存的问题。

程序也会出错，比如写入文件时，如果磁盘空间不足写入操作就会失败。再如，用煤气灶烧开水时，如果火被风吹灭，煤气将充满房屋。如果煤气泄漏检测仪没有检测到这一险情并报警的话，后果将不堪设想，或将酿成煤气爆炸这样的大灾难。[①]

程序的出错也一样。写入文件失败时，如果没有任何警告，用户就难以察觉。用户可能会误以为这些非常重要的数据已经成功写入磁盘中，从而把原始数据删除。为了能使用户在灾难发生前有所警觉，程序设计语言需要具备和煤气泄漏检测仪一样的错误传达机制。

① 当然，运气好的话可能会闻到煤气气味而不至于酿成大祸，但这毕竟是靠运气。现在城市供应的煤气中通常会特意加入硫化氢等有恶臭味的添加物，就是为了让大家能尽早发现煤气泄漏，尽量减少对运气的依赖，从而构建起双重安全保障。

6.2 如何传达错误

假设有一个执行时可能出错的函数，取名为 shippai。调用这个函数时，有可能成功也有可能出错。那么该如何分别编写成功时的操作和出错时的操作（错误处理）呢?

错误处理的编写方法大体可分为两种。一种是利用 shippai 函数的返回值来传达程序出错的信息，函数调用方通过检查返回值来相应地对错误进行处理。另一种是在调用 shippai 函数前设定好错误处理的代码，错误发生时能跳转至相应的错误处理代码。前者至今还在 C 语言等语言中经常使用，后者则被称为异常处理。

通过返回值传达出错信息

我们首先来看通过返回值传达出错信息的方法。比如，shippai 函数事先将返回值定义为成功执行时返回 0 值，出错时返回 0 以外的值。函数调用方在调用 shippai 函数后，会检查其返回值，返回非 0 值时会进行错误处理。

C语言
```
if(!shippai()){
  /* 错误处理 */
}
```

※ !x 即为 x 非 0 。

这也就是说，出错时把信息写入返回值，接着做返回值检查。与之相类似的有先行定义全局变量接收错误信息的方法，还有在函数调用方定义变量作为引用形参接收错误信息的方法。无论哪种，遵循的都是出错时写入信息然后做检查的思路。

这种方法在 C 语言等众多语言中广泛使用。但是它有两个问题：

- 遗漏错误
- 错误处理导致代码可读性下降

下面我们来详细讨论这两个问题。

遗漏错误

首先是遗漏错误，程序员忘记了对返回值做检查，从而遗漏了错误。

程序员也是普通人，他们也会时常忘记 shippai 函数有可能会出错。如此一来，shippai 函数调用完后没有对返回值做检查，按照执行成功的固定思维编写了代码[①]。

如果 shippai 函数是一个极少执行出错的函数，那这段代码在大多数情况下都可以正确运行，程序员就会深信这段代码没有问题。发现有问题时，正是 shippai 函数执行出错，程序执行与预期不一致的时候。谁都不知道这将是什么时候的事了，或许是在产品正式发布了之后。

这种情况下，由于编写代码和发现问题在时间上已经相差较远，代码问题的追查往往会很辛苦。此外，由于 shippai 函数执行出错，某个值可能与期待值不一样，进而导致别的函数执行出错，如此下去将形成多米诺骨牌式的连锁反应。问题是在看起来和 shippai 函数没有任何关系的地方被发现的，这样一来就更难发现真正的问题所在了。

理想的情况是，程序员在准备调用函数时，先确认好该函数是否可能执行出错以及出错时返回什么值。如果能严格按此操作进行，使用返回值传达错误的方法应该不会造成遗漏错误的情况。然而，现实中因忘记检查返回值导致错误的情况不胜枚举。

那该怎么办呢？

错误处理导致代码可读性下降

接下来，我们来看第二个问题。知道了第一个问题，对所有的函数调用都做仔细的检查以免遗漏错误，然后着手编写错误处理的代码，然而接踵而来的是另一个问题。由于错误处理，源代码变得很难读懂。

执行三个可能出错的操作，如果在某处操作失败，接下来的操作就不会被执行转而进行错误处理。这里的三个操作依次是 shippai("A")、shippai("B")、shippai("C")，程序代码如下：

---

① 比如，C 语言中写入文件操作时常用到 fprintf 这个函数，它在执行出错时返回值为负。你对这个有做检查吗？还是想当然以为执行成功从而遗漏了错误呢？

C语言
```
int main(){
  if(!shippai("A")){
    /* 失败时的处理 */
    /* 失败时的处理 */
    /* 失败时的处理 */
  }else if(!shippai("B")){
    /* 失败时的处理 */
    /* 失败时的处理 */
    /* 失败时的处理 */
  }else if(!shippai("C")){
    /* 失败时的处理 */
    /* 失败时的处理 */
    /* 失败时的处理 */
  }
}
```

本来想要执行三个操作，而在代码实现中引入了大量错误处理代码，夹在本意要执行的操作中，这使程序流变得难以读懂。

那么就没有可读性更好的编写方法吗？如果错误处理相同[①]，能不能集中处理？怎么样才能集中起来呢？

通过跳转集中进行错误处理

为了集中进行错误处理而使用异常，但 C 语言里并没有异常这一机制。它使用 goto 语句。下面的代码把错误发生时的处理用 goto 语句集中起来[②]。

C语言
```
int main(){
  if(!shippai("A")) goto ERROR;
  if(!shippai("B")) goto ERROR;
  if(!shippai("C")) goto ERROR;
```

① 比如，打开日志文件，增加标识执行失败的信息，然后关闭日志文件这样的错误处理。

② Linux 的发明者林纳斯·托瓦兹（Linus Torvalds）在其文章“Linux 内核编码风格”中推荐使用 goto 语句把函数的结尾处理集中起来。英文版可参照：http://www.linuxfromscratch.org/alfs/view/hacker/part2/hacker/coding-style.html。

```
  return;
ERROR:
  /* 失败时的处理 */
  /* 失败时的处理 */
  /* 失败时的处理 */
}
```

我们来看一下这段代码，执行 shippai("A") 如果出错，跳转至 ERROR。成功则移至下一行执行 shippai("B")，如果出错跳转至 ERROR。同样执行 shippai("C")，如果出错，跳转至 ERROR。如果都成功则执行下一行 return，返回跳出函数。这样，错误处理只在跳转至 ERROR 处时才被执行。

从代码形式上看，这就做到了把针对出错时的代码和记述本来想做的事的代码分离。

出错则跳转

到此为止，我们学习了传达错误的方法之一——通过返回值传达错误。现今，这一方法在以 C 语言等很多语言中被广泛使用。

实际上，在 C 语言诞生以前就已经存在其他错误处理方法。这种方法事先定义好了错误发生时跳转的位置，后来，它演变为现在的异常处理。为更好理解地异常处理产生的原因，我们来回顾一下那段历史。

UNIVACI

事实上，错误发生时跳转这一想法的产生甚至比程序设计语言的产生还要早。1950 年设计的计算机 UNIVACI 中就有了这样的功能，在计算中出现溢出时，它会执行在 000 处编写的命令。这种功能被称为“中断”（interrupt），广泛被运用于错误处理等各领域中。比如，键盘上某按键被按下时，CPU 就能收到按键信号，传达这一消息的就是中断功能。

COBOL

早期的程序设计语言是如何进行错误处理的呢？ 1954 年出现的程

序设计语言 FORTRAN 语言中还没有异常处理机制，直到 1959 年，新问世的 COBOL 语言中才设计了两种类型的错误处理机制。与现代异常处理中的通用语句结构不一样的是，这两种类型的错误处理都有各自独特的语句结构。一种结构是用 READ 命令读取文件时，由 AT END 关键字引出没有数据等错误处理的语句。另一种结构是用 ADD 命令做数值的四则运算时，由 ON SIZE 关键字引出溢出等错误处理的语句。

```
COBOL
READ <文件名> AT END <错误处理语句>
ADD <函数名> ON SIZE ERROR <错误处理语句>
```

当时错误处理仅有这两种类型，程序员无法自由增加设计错误的种类，这和现代的异常处理机制是不同的[①]。

PL/I

到了 1964 年，PL/I 程序设计语言诞生了，它是 FORTRAN 语言、COBOL 语言、ALGOL 语言的集大成[②]，为实现灵活统一的错误处理，引入了 ON 语句结构。当时有的 PL/I 教材中列举 GOTO、IF、DO 和 ON 这四种语句来讲解控制语句，它们分别对应跳转语句、条件语句、循环语句和异常处理[③]。

与 6.2.1 节介绍的 C 语言代码相近，下面的代码是在 PL/I 语言中的表现方式。

```
PL/I
SHORI: procedure;
  on error go to ERROR;    /*出错时跳转至ERROR处的意思*/
  call shippai(1);
  call shippai(2);
  call shippai(3);
  return;
```

---

① 《情報処理月例会資料》（中文译名：信息处理每月例会资料）1965/8/24

② 《プログラミング言語の歴史と展望》（中文译名：程序设计语言的历史与展望），中田育男，《情報処理》（中文译名：信息处理）Vol.21, No5, 1980 年，p.574。

③ 《わかりやすいプログラミング 4》（中文译名：简单易懂的程序设计 4 PL/I），竹下亨，1969 年。

```
  ERROR: <失败时的处理>;
  <失败时的处理>;
  <失败时的处理>;
end;
```

C 语言代码中通过使用 if 语句来检查返回值，这里已经没有检查操作。这里体现的思想是，不是让程序设计者编写程序时，时刻记着不能忘记返回值检查，而是让语言处理器来自动检查是否出错。

另外，在 COBOL 语言中仅有的两种错误类型的基础上，PL/I 中可以追加定义新的错误类型，从而可以根据错误类型的不同，轻松地变换错误处理的操作。

```
PL/I
/* 定义名为MY_ERROR的条件 */
dcl MY_ERROR condition;

on condition (MY_ERROR)
  begin;
    /* MY_ERROR发生时的错误处理 */
  end;
```

PL/I 语言的错误处理机制还有另一种重要功能，程序可以主动触发新定义的错误类型。在 C 语言通过返回值传达错误的方法中，如果是 return –1，就会返回 –1 向调用处传达出错信息。而在 PL/I 语言中，通过语句 signal condition (MY_ERROR)，可以触发 MY_ERROR，传达出错信息。

```
PL/I
/* 触发MY_ERROR */
signal condition (MY_ERROR);
```



可追加错误类型和可自主触发出错，这两种功能为现代的异常处理机制所继承，具有重要意义[①]。

① 当时不叫失败或异常，叫条件（condition）。另外，抛出异常的命令也不是现在一般的 raise 或 throw，而是 signal。

6.3 将可能出错的代码括起来的语句结构

至此我们了解到，到 1964 年 PL/I 语言诞生时，很多对当今的异常处理意义重大的特征已经被提出来了，如允许定义出错时的处理操作，可以追加新的错误类型，可以自主触发出错等。

然而，它和现在 Java 语言、C++ 语言、Python 语言等采用的异常处理的语句结构有很大的不同。PL/I 语言是先定义好出错时的处理操作，再编写可能出错的代码。与这种形式不同的是，Java 等语言是先（用 try{...} 括起来）编写可能出错的代码，然后编写出错时的处理操作。那么这种语句结构是何时、基于什么原因产生的呢？

John Goodenough 的观点

1975 年，John Goodenough 在自己的论文[①]中提出了一种更好的异常处理的方法[②]。他的观点是这样的：命令有可能会抛出异常，而程序员有可能忘记这种可能性，也可能在不正确的地方编写异常处理或者编写不正确类型的异常处理。为使编译器能够对程序员的错误发出警告，减少这种可能性，需要做到两点。一是明确声明命令可能抛出何种异常，二是需要有将可能出错的操作括起来的语句结构。

以这里提议的括起来的语句为基础，现代大部分语言采用了先括起来可能出错的操作，再编写错误处理的语句结构。明确声明命令可能抛出何种异常，这个设计方针在 Java 语言的异常检查中得以继承，我们将在后面的 6.6 节中详细解说。

---

① John B. Goodenough, “Exception handling: issues and a proposed notation”, *Communications of the ACM*, Vol.18 Issue 12, ACM, 1975, pp.683-696 此时把失败称为异常（exception）已经很普遍了。

② John Goodenough 在创作这篇论文时是 SofTech, Inc 的职员，之后担任卡耐基梅隆大学软件工程研究所的最高技术负责人。更多详细资料请参考：http://www.sei.cmu.edu/about/people/profile.cfm?id=goodenough_12984

引入 CLU 语言

从 1975 年 Goodenough 的论文发表直到 1977 年，程序设计语言 CLU[①] 引入了异常处理的机制，追加了置于命令后面的错误处理语句结构 except。CLU 语言从最初就具有用 begin... end 将代码括成块状的功能，这一功能和 except 相结合，就实现了将可能出错的操作括起来再补充错误处理的代码编写方式。以下是一段 CLU 语言代码，从百分号到句末为注释。错误的类型，可以是诸如除数为零时的 zero_division[②]。

```
CLU
begin
  % 可能出错的操作
  % 可能出错的操作
end except when 错误的类型:
  % 出错时的处理
  % 出错时的处理
end
```

引入 C++ 语言

不久后的 1983 年，C++ 语言诞生。针对异常处理的语句结构问题从 1984 年到 1989 年间经历了多次讨论，C++ 语言最终确认追加一种语句结构，把关键字 try 放在那些被括起来的可能出错代码的前面，把关键字 catch 放在捕捉并处理错误的代码块前面[③]。按照 C++ 语言设计者斯特劳斯特卢普（Bjarne Stroustrup）的说法，try 只是一个为了方便理解的修饰符[④]。

---

① 后文介绍 Liskov 置换原则的章节中会提到 CLU 语言的发明者——Barbara Liskov，请参考 12.1 节。

② CLU 语言中抛出异常的命令和 PL/I 语言一样，都为 signal。

③ 关于这个过程的详细描述，请参考斯特劳斯特卢普的著作 *The Design and Evolution of C++*。BASIC 语言和 PL/I 语言具有的错误处理后返回出错那行的 resume 功能，该功能被取消的过程颇为有趣，推荐读者阅读。

④ “try 也不是必需的，但没有它理解不便，因此最终决定引入这一看起来多余的关键字”。请参考斯特劳斯特卢普 的著作 *The Design and Evolution of C++*。

C++
```
try {
  /* 可能出错的代码 */
  /* 可能出错的代码 */
} catch {
  /* 错误处理命令 */
  /* 错误处理命令 */
}
```

另外，C++ 语言还选用 throw 作为触发异常的命令[①]。之所以没选择 PL/I 语言和 CLU 语言中使用的 signal，是因为 signal 已经在标准库中被使用了。于是，触发异常的表述就变成了抛出异常。

引入 Windows NT 3.1

上世纪 90 年代初，微软公司开始用 C 语言编写新的 Windows 操作系统，这就是 1993 年发布的 Windows NT 3.1。这个版本的制作时，也考虑到需要有便于操作的错误处理机制，于是在操作系统和 C 语言编译器导入了结构化异常处理（Structured Exception Handling，SEH）的概念。结构化异常处理中，除了将可能出错的代码括起来的 __try 和将错误处理的代码括起来的 __except 之外，还有将即使出错也要执行的代码括起来的 __finally。结合随后出现的语言中的叫法，我们把表示即使出错也要执行的关键字称为 finally。下一节我们来讲 finally 的必要性。

C语言中使用SHE的代码
```
__try{
  __try{
    /* 可能出错的代码 */
    /* 可能出错的代码 */
  }__finally{
    /* 出错与不出错都要执行的代码 */
```

① “使用 throw 这一关键字是因为更易理解的 raise 和 singal 两个关键字已经在标准库作为函数名字占用了”。请参考斯特劳斯特卢普的著作 *The Design and Evolution of C++*。

```
  }
}__except(...){
  /* 错误处理的代码 */
}
```

6.4 出口只要一个

为什么引入 finally

微软公司为什么会引入 finally 呢？是为了解决什么样的问题呢？他们是这样回答的。

> 采用结构化异常处理可以提高代码的可靠性。比如，程序在程序员预料之外结束时，也可以正确地释放锁定的内存和文件等资源。另外，针对内存不足等特定问题时，不需要使用 goto 语句或细致地检查返回值，使用简洁的结构化代码就可以应对。
>
> ——“异常处理（SEH）”
> http://msdn.microsoft.com/ja-jp/library/aa984822(v=vs.71).aspx

成对操作的无遗漏执行

在程序设计中有很多成对的操作，比如内存锁定后要释放，文件打开后要关闭，上锁之后要解锁等。成对的操作只要执行了其中一个，就要保证另一个也能确实执行。微软公司前面提到的“正确地释放资源”，指的就是无遗漏地执行成对操作。对错误处理，则要能够不使用返回值检查和 goto 语句，简洁地实现。

拿美术馆来打个比方吧。在入口借出的语音导航仪随后需要全部回收。只有一个出口的话，全部回收并非难事。但是如果有多个出口，不

在每一个出口都配备专人的话，就不一定能做到无遗漏全部回收。如果连墙壁都没有，随处都可以出去的话，要无遗漏地全部回收就更困难了。

程序设计也一样。如果在入口处执行了 lock，随后就需要执行 unlock。如果函数出口（return）只有一个，只要在出口前执行 unlock 即可。但如果有好几个 return，就需要在每一个 return 前面执行 unlock。一个被调用的函数，如果在多处都有抛出异常的可能性，那么在很多个地方都有可能跳出这个函数，此时要无遗漏地执行 unlock 就变得非常困难了。

C语言

```
lock(m);
/* 需要上锁的处理 */
if(...){
  unlock(m); /* ←在出口前解锁 */
  return;
}
/* 需要上锁的处理 */
unlock(m); /* ←在出口前解锁 */
return;
```

这个问题的解决办法有三种。

使用 finally 的解决方案

finally 代码块在 try 代码块执行结束后一定会被执行到，而不管 try 代码块中是否发生异常。1990 年左右，微软公司开始使用 finally，1995 年发布的 Java 语言也引入了 finally。现今，Python 语言和 Ruby 语言也支持同样的语句结构[①]。

Java

```
try{
  /* 可能出错的代码 */
} catch(...) {
  /* 异常处理代码 */
} finally {
```

① 顺便说一下，Python 语言中还有 else 小节，这些代码在 try 小节成功执行后能继续被执行并且不需要捕捉可能的异常。

```
  /* 必将执行的代码 */
}
```

Ruby

```
begin
  # 可能出错的代码
rescue
  # 可能出错的代码
ensure
  # 可能出错的代码
end
```

Python

```
try:
  # 可能出错的代码
except:
  # 异常处理代码
finally:
  # 必将执行的代码
```

没有 finally 的 C++ 语言的解决方案

C++ 语言中没有 finally。那它是如何表现不管异常是否发生都要执行的代码的呢？

C++ 语言中使用了一种名叫 RAII（Resource Acquisition Is Initialization，资源获取即初始化）的技术。比如，在操作打开了就要关闭的文件对象时，定义来操作该对象的类，用构造函数打开，用析构函数关闭[①]。

C++

```
class SampleRAII {
public:
  // 构造函数
  SampleRAII()
    : resource(lock()) {
  }
```

① 不习惯 C++ 语言的读者可以这样理解：简单来讲，构造函数和析构函数就是对象在被创建时调用的初始化处理和在消除时调用的事后处理。

```
  // 析构函数
  ~SampleRAII() {
    unlock();
  }
……
}
```

这种技术在函数结束时，针对函数的局部变量，程序可以自动地调用析构函数[①]。C++ 语言设计者斯特劳斯特卢普认为，比起使用 finally，这种方法更为优雅。

D 语言中 scope (exit) 的解决方案

2001 年出现的 D 语言以改良 C++ 语言为目标，反对 RAII 是优雅的这一意见。

打开了就要关闭这样紧密关联的操作，反映在代码上时，如果能放在相近的位置就容易理解多了。基于这一考虑，D 语言中引入了作用域守护（scope guard）的概念。通过使用作用域守护，可以事先定义从某一作用域（如函数）跳出时执行的操作。

```
D语言
void abc()
{
    Mutex m = new Mutex;

    lock(m); // 锁住mutex
    scope(exit) unlock(m);    // 定义作用域结束时的解锁操作

    foo();    // 执行操作
}
```

① 严格来讲，不是函数而是空间范围，为避免 C++ 语言里复杂的空间范围的话题，此处做了简单化处理。另外，实际上 lock 和 unlock 是带有参数的，代码会比这更复杂，这一点也进行了简单化处理。

6.5 何时抛出异常

到此为止，我们学习了 try/catch 括起来的异常处理结构语句是怎样产生和发展的，主要围绕异常被抛出来之后如何处理进行了解说。接下来我们要转移一下焦点，来学习异常是什么时候抛出来的。

错误发生时，有返回返回值和抛出异常两种传达方法。那么，什么时候使用返回值的方法，什么时候使用异常的方法呢？ 2000 年左右，有种观点认为，异常的方法仅限于异常的情况下使用 ①，那么异常的情况又是指哪些情况呢？

函数调用时参数不足的情况

这里我们列举 Python、Ruby、JavaScript 这三种脚本语言，来比较各种语言分别在什么时候抛出异常。比如，调用一个带有两个参数的函数但只传递一个参数时会发生什么？ Python 语言和 Ruby 语言会在函数调用的时刻抛出异常。但是 JavaScript 语言会把缺失的参数当作未定义的特殊值（undefined）继续执行。

```
Python
def foo(x, y):
  print x, y

foo(1)

结果（异常）
Traceback (most recent call last):
  File "tmp.py", line 4, in <module>
    foo(1)
TypeError: foo() takes exactly 2 arguments (1 given)

Ruby
def foo(x, y)
```

① 参照 http://www.ibm.com/developerworks/jp/java/library/j-perf02104/index.html

```
  p x, y
end

foo 1
```

结果（异常）

```
tmp.rb:1:in `foo': wrong number of arguments (1 for 2) (ArgumentError)
        from tmp.rb:5:in `<main>'
```

JavaScript

```
function foo(x, y){
  console.log(x, y);
}

foo(1)
```

结果（成功）

```
1 undefined
```

数组越界的情况

还有一种情况，比如，试图读取一个只有三个数的数组的第四个数值时会怎么样？这就是数组的界外操作。此时，Python 语言会抛出异常，Ruby 语言会返回一个指示不存在的特殊值（nil），而 JavaScript 语言会返回 undefined。

Python

```
x = [0, 1, 2]
print x[3]
```

结果（异常）

```
Traceback (most recent call last):
  File "tmp.py", line 2, in <module>
    print x[3]
IndexError: list index out of range
```

Ruby

```
x = [0, 1, 2]
```

```
p x[3]

结果（成功）
nil

JavaScript
x = [0, 1, 2];
console.log(x[3]);

结果（成功）
undefined
```

以上三种语言的设计者都是具有高超技术能力的程序设计员，即便是他们，就何种情况应该抛出异常也不能达成一致。异常应该在何种情况下使用，何为异常的情况，这些问题是没有正确答案的。

出错后就要立刻抛出异常

笔者认为，在学习程序设计或者是一个人编写小规模程序时，像Python语言这种立刻抛出异常的方式要比JavaScript语言那样返回undefined更好。

人非圣贤，孰能无过，程序员也不例外，一不小心引入bug也不是不可能的。要保证代码的品质只能是尽早发现bug并及时修正它。这同时说明，能尽早意识到不对劲的地方是十分重要的[①]。

异常机制的优点就在于，它不会遗漏任何一处错误。当发生数组的界外读取时，我们希望程序能抛出异常，向程序员报告错误。只要不是程序员有意编写“界外读取时返回undefined”这样的代码，界外读取的行为终究是异常事态。返回适当的值来推迟宣布异常事态的发生，这样也无法解决异常[②]。

① 为了提高软件的品质通过检查代码来确认程序员期待的程序行为，这一测评方法也是基于同样的想法。

② 因为没有注意到返回值为undefined而继续做操作，接下来会碰到“TypeError: Cannot read property 'foo' of undeinfed”，这个对于JavaScript的程序员应该是家常便饭了吧。

发生错误应该停止操作立刻报告，这一设计思想被称为错误优先（fail first）。软件的目的不一样，简单地停止操作有时可能不太妥当，但至少在学习和开发阶段发生错误后能立刻注意到，这已经是一个很大的优点了。

6.6 异常传递

包括 Java 语言在内的很多现代语言的异常处理机制中，异常可传递至调用方。假设函数 f 调用函数 g，后者又调用函数 h。如果函数 h 中有异常抛出且在函数 h 中无法处理该异常，那么就会看函数 g 能否处理该异常。如果函数 g 也无法处理，接下来就看函数 f 可不可以。如果没有哪个函数可以处理该异常，程序就会异常终止。

异常传递的问题

大家或许认为这样做是理所当然的。其实关于这个意见并未统一，因为这一设计有一个很大的问题。那就是，即使看到了函数 f 的代码也不知道函数 f 可能会抛出什么异常。有可能是函数 f 调用的另外的函数 g 中抛出的异常传递过来的，也有可能是函数 g 调用的函数 h 抛出的异常。也就是说，如果不看见函数 f 调用的所有的函数代码，就无从得知函数 f 抛出何种异常。万一没有察觉到抛出某种异常的可能性，程序就有可能异常终止。

Java 语言的检查型异常

在 6.3.1 节提到的论文中，Goodenough 主张为了避免这一问题，需要明确地声明可能抛出的异常。Java 语言就采用了这一方针，我们现在就来看一下。

其他语言中所谓的异常，Java 语言中的 throw 语句也能抛出，并进

一步分为三类：不应该做异常处理的重大问题、可做异常处理的运行时异常和可做异常处理的其他异常。这里的其他异常叫做检查型异常，如果在方法之外抛出，就需要在定义方法时声明。

throws 就是为这个目的准备的。以下代码中的 void shippai ( ) throws MyException，实质上是声明了这个方法有可能抛出 MyException 的异常。

```java
class Foo {
    // shippai抛出MyException异常
    void shippai() throws MyException{
        throw new MyException();
    }

    // 使用shippai（方法1）声明'throws MyException'、
    void foo() throws MyException{
        shippai();
    }

    // （方法2）用catch捕捉MyException异常、错误处理
    void bar(){
        try{
            shippai();
        }catch(MyException e){
            ...
        }
    }
}

class MyException extends Exception {}
```

使用了检查型异常，就不会发生漏查抛出异常的可能性。当一个方法调用可能抛出异常的方法 shippai 时，可以选择两种方法来实现异常处理，将 shippai 抛出的异常传递至调用处，或者让 shippai 自己处理该异常。前者就如方法 foo 那样用 throws 声明，后者就如方法 bar 那样用 try/catch 括起来的语句实现异常处理。如果没有采用两者中任何一种方法，编译器就会指示出遗漏错误。

检查型异常没有得到普及的原因

可以说检查型异常是一种非常好的机制。但是这种机制并没有很好地普及到其他语言中，这是为什么呢？

一言以蔽之，就是因为它太麻烦。一旦 throws 或 try/catch 中异常的数目增多，或者某一方法需要追加一种异常，就不得不修改调用了该方法的所有方法，特别麻烦。

C# 语言虽然很大程度上参考了 Java 语言，但也没有采用检查型异常机制。究其原因，C# 语言的设计者 Anders Hejlsberg 如是说："检查型异常的理念本身没有什么不对的地方，相反是很棒的。然而在像 Java 语言这样的实现方式下，它在解决某些问题的同时又引入了别的问题。如果能有更好的实现方法，C# 语言或许也是可以借用的[①]。"

专栏

具体的知识和抽象的知识

在语言 X 中如何实现 Y，像这种具体的知识（know-how）可快速提高你的工作效率。但是一旦语言发生变化，这种知识就无法再使用。世界瞬息万变，这意味着限定了应用范围的具体知识将慢慢失去其价值。因此，我们不仅要学习具体的知识，更要有意识地去学习那些应用范围广泛的抽象的概念。

当然，学习了抽象的元知识，如果不将其与你具体的经验相结合，也无法在实际应用中发挥其作用。喜欢樱花的人即使剪下花开的树枝带回家，终将看到的也仅仅是枝枯花败的场景而已。要想樱花年年盛开，离开根部和枝干是不行的。

我们所学的知识到底有没有真正的"根基"，可以通过考察能否具体地举例或者具体地实现来确认。没有真正根基的知识是无法顺藤摸瓜、触类旁通的，所谓学习到的知识也只能像鹦鹉学舌般的重复讲讲而已。想要因地制宜地活用知识更是缘木求鱼，根本没有可能了。

① 请参考"The Trouble with Checked Exceptions"，http://www.artima.com/intv/handcuffs.html

专栏

学习讲求细嚼慢咽

一口吞不下一整块肉。首先要把肉切成能入口的大小，嚼碎后再吃。同样的道理，对抽象的概念、复杂的系统和不习惯的领域，我们也不可能一下子理解通透。首先要把信息切分，一小块一小块地消化吸收到自己的大脑里。

然而，在信息爆炸时代，如何对信息做取舍呢？什么信息是重要的，什么是不重要的？要判断什么信息重要首先需要对其有深刻的理解，但如此一来，就陷入到先有蛋还是先有鸡的困境中了。

身边如果有这熟悉这些信息的人、朋友，向他们请教也是一种方法。但要是没有呢？在网上检索查询的话，那些发言者是真的熟悉还是装作熟悉呢，这又该如何判断？

作者本人写的文档当然是最为翔实的。但是要么认为难懂，要么认为内容太多，要么认为看英语有困难，我们总会有各种借口放弃查阅原作者的一手资料，而去寻找那些写得简单的解说。这就如同，当肉太大、太硬时，转而不顾食品安全，吃起别人做出来的肉馅。

这种心情是可以理解的。笔者也有在庞大信息量面前心力交瘁的时候。这时有三种战略可供参考：从需要的地方开始阅读，先掌握概要再阅读细节，从头开始逐章手抄。关于它们的详细介绍将在接下来的章节介绍。

6.7 小结

本章我们学习了程序也会出错以及程序出错后是如何传达错误的。错误传达方法大致有两种，通过返回值传达和出错后跳转。

通过返回值传达的方法有一个问题，那就是容易忘记检查返回值从而出错。因此，长期以来，人们把更多的精力放在研究出错后跳转的方法上，这一方法成为目前被 Java、C++、Python、Ruby 等众多语言支持的异常处理机制的基础。

异常处理也有几个问题。一个是当函数有不只一个出口时，必须成对处理的操作很难正确地成对处理。另一个是即便看了代码也不知道函数将抛出何种异常。

2012 年，谷歌公司在其编程守则中禁止了 C++ 语言中异常的使用。

> 使用异常的优势大于它的成本，尤其是在新的项目中。但是，在已有的代码中导入异常时，有依存关系的所有代码都会受到影响。（中略）谷歌公司已有的 C++ 语言代码大多不是按异常处理实现的。因此，引入触发异常处理异常的新代码多少是有些困难的。
>
> ——《Google C++ 风格指南》[①]
> http://www.textdrop.net/google-styleguide-ja/cppguide.xml

谷歌公司一方面认可在新的项目中使用异常是好的，另一方面考虑到现在已经有很多不是基于异常处理实现的代码，不能使用异常。这是因为，导入异常后要成对地处理那些必须成对的操作非常困难。

Java 语言的开发者为了解决第二个问题导入了检查型异常，但是这种方法并不太被接受。C# 语言的开发者一方面承认检查型异常的优势，另一方面希望有更好的方法出现。

最后的情况如我们所见，两种错误传达方法都是长处短处兼具。把握各自的优势和缺点，因地制宜地选择使用哪一种才是最为关键的。

专栏

从需要的地方开始阅读

针对面对庞大信息量心力交瘁时该怎么办的问题，我们在“学习讲求细嚼慢咽”专栏中介绍了三种方法。第一种方法就是“从需要的地方开始阅读”。

有时，我们并不需要掌握一本书或者文档的全部内容。明确阅读的目的并弄清楚为达成这一目的需要阅读哪些地方，就可以有针对性地阅读，无需在其他无关的地方花费大量精力。

或许有些人会因为不逐字逐句仔细阅读而产生负罪感。但是比起心力交

① 英文最新版请参考：http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

瘁完全读不下去来说，还是挑挑拣拣地阅读更好吧。与其让“非全部读完不可”这样的完美主义挫败干劲，还不如放弃不读呢。要知道，干劲是多么地宝贵呀！

使用这一方法就需要对整体内容有个大致把握，并确认想阅读的部分。如果不知道如何把握整体内容，请参考接下来的专栏“先掌握概要再阅读细节”（第 8 章），试试里面介绍的方法吧！

第7章

名字和作用域

变量和函数都有名字。
名字是如何产生的呢?
本章将阐述名字和作用域。
作用域是指名字的有效范围。
那么为什么要对名字的有效范围作出限制呢?
本章我们将回顾作用域的演变历史，学习名字和作用域有哪些好处。

7.1 为什么要取名

程序设计中，名字的发明至少在 50 年前。给变量和函数取了合适的名字后，程序的可读性显著提高。由于大部分语言都在使用名字，现在看来，取名似乎是理所当然的事了。

那么名字是缘何发明的呢？为解答这个问题，我们反过来想一下。在名字发明以前，程序员是如何指示现在那些用名字指示的内容的呢?

答案是使用编号[①]。计算机记录数据的存储位置可以用编号来替代，于是有了“3456 号的数值加 1”这样的计算机指令。

然而，人们都会觉得给内容或位置取个易于理解的名字，并用该名字来指示它们会更加方便。比如，从书架上取书时，相比使用“取出 3456 号书”或者“取出 ISBN 编号为 978-4-8399-2282-5 的书”来指定，“取出《Python 程序设计》这本书”这样直接使用名字的指定方法要方便得多。相比“把 12345 号放置的内容移到 98765 号位置”，“把桌子上的信封投入家门口的邮筒中”这种说法简洁明了得多[②]。因此，程序设计

① 也叫做存储地址，即指针变量中的内容。

② 同样地，相比“74.125.235.164 的服务器”来说，“google.com”更易于理解，故使用 DNS。相比“纬度 35.693449”来说“东京都新宿区某街”更容易理解，故使用居住地址。

语言也逐渐演变为使用名字来指定对象了。

怎样取名

怎么把“取出书架左边第 36 本书”的说法转换成“取出《广辞苑》”呢？如果计算机有名字和内容的对照表就可以了。

```
{广辞苑 => 第36本，大辞林 => 第37本，新明解国语辞典 => 第38本}
```

有了上面这样的对照表，计算机在接到“取出《广辞苑》”的指令时，就可以理解成“原来说的是第 36 本书”。

用 Ruby 语言代码来试着创建 x、y、z 三个变量，如下所示：

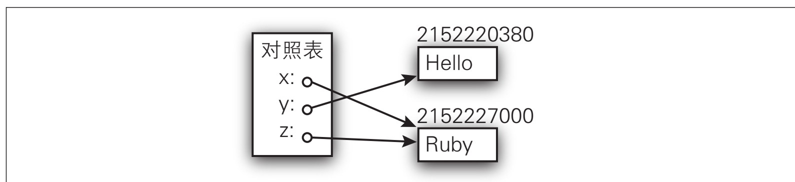
```ruby
x = "Ruby"
p x.object_id #=> 2152227000

y = "Hello"
p y.object_id #=> 2152220380

z = x
p z.object_id #=> 2152227000
```

P 相当于其他语言中的 print，其输出结果写在注释符 #=> 的右边。如图 7.1 所示，x 和 z 这两个名字是与 2152227000 号的内容相对应的，而 y 这个名字是与 2152220380 号的内容相对应的。

图 7.1　名字与内容（对象）的对照表



因为 x 和 z 指示同样的内容，将 x 的开头改写为 P 的话，z 也会变成 Puby[①]

① 这样讲可能有些跑题，将这个例子理解为“变量就是盒子”的话，有些费解吧。

```
Ruby
x[0] = "P"
p x #=> "Puby"
p z #=> "Puby"
```

名字冲突

早期的程序设计语言中，对照表是整个程序所共有的，就好比一个公司里有一块大的白板，大家都在上面做记录一样。这里就会有一个很大的问题，我们举例来说明。

下面的代码在变量 i 的值从 0 逐增到 10 的过程中调用函数 shori[①]。

```
Perl
for($i = 0; $i < 10; $i++){
    &shori();
    print "处理", $i, "结止\n";
}
```

初看这段代码的人，可能会觉得这里的 for 语句在做完 10 次处理后会终止。其实并非一定如此。

如果在函数 shori 中不小心使用了 i 这个名字会怎样呢？这时 i 的值就被改写了。下面的例子中，每次函数 shori 被调用时变量 i 的值都被重置为 0，for 语句不管循环多少次，变量 i 都不会变为 10。于是就变成了无限循环。

```
Perl
sub shori{
    $i = 0;
}
```

如何避免冲突

前面提到的“整个程序共用一个对照表”，也就是说，“变量名字的

① 从这里开始，我们会时不时使用 Perl 语言的代码来举例说明，这是因为 Perl 语言是可以使用很多种作用域的语言。

有效范围是整个程序”。这种情况也可以称为“该变量具有全局作用域”，讲得更简洁一些，“这是一个全局变量”。全局变量每次被改写它的影响会波及整个程序。因此，在函数实现时不得不考虑其调用处哪个名字的变量会被改写。上面的例子中，因为函数 shori 中使用了调用处同样使用了的变量 i，程序就陷入了无限循环之中。

要防止变量名不小心重复使用，即要防止名字冲突，该怎么做呢?

取更长的变量名

一种方法就是取更长的变量名。shori 函数中不要使用 i 这样短的名字，而是都换成 i_in_shori 这样的名字，之后名字冲突就不可能发生了吧。也有一些其他方法，比如，在多人共同开发的项目中，采用变量名使用申请制度，或者在变量名中加入开发者的名字，或者在变量名中加入连续的编号使之不重复。

使用作用域

随着程序规模的扩大，防止名字冲突的考量也变得艰难起来。没有人会觉得管理变量的名字是件开心的事情。没有其他更好的方法了吗？于是另一种使用作用域的概念的方法产生了。下一节我们来学习作用域。

7.2 作用域的演变

作用域是指名字的有效范围。要保证程序整体不会出现名字冲突是件困难的事情，为此将名字的有效范围限定在更小的范围之内，让程序管理变得轻松一些。作用域的提出就是基于这个考虑。前面讲到的问题点，就是在函数 shori 中名字 i 所赋有的值被改写的情况下，会给其他部分的程序带来影响。如果函数 shori 中的名字 i 仅在此函数中有效就好了。这样一来，就只需关注 i 在函数 shori 中的情况就可以了。那么，该如何实现这一点呢?

动态作用域

如何操作

一种解决方法是，把变量原来的值事先保存在函数入口处，在出口处写回变量中。

```
Perl
sub shori{
    $old_i = $i; ❶
    $i = 0;  # 各种处理
    $i = $old_i; ❷
}
```

函数的开头把变量 i 的值代入变量 old_i 中（❶），函数结束时再把变量 old_i 中的值返回给变量 i（❷。这是一种先把原来的值另存起来随后返回的方法。如此一来，在❶和❷之间不管怎样改写 $i 的值，对其他部分都没有任何影响。

使用这种方法，必须要注意不要漏写返回原来的值的那一行❷。函数中有 return 返回时，也要记得先把变量原来的值返回后再执行 return 语句。凡当函数退出时，所有地方都要毫无遗漏地加上返回值的代码[①]。

然而人总是会犯错误的，对于这样的任务，我们总是希望能让计算机去完成。从 1991 年发布的 Perl4 开始，Perl 语言 增加了这一功能，通过把变量声明为 local 就可以让程序处理器去承担“把原来的值另存起来随后返回”的任务。

```
Perl
sub shori{
    local $i;
    $i = 0;
}
```

这样的作用域称为动态作用域。

① 从函数中退出时保证无遗漏地执行某些特定的操作，这个和异常处理结合在一起时，会变成一个复杂的话题。关于这一点，第 6 章有详细的介绍。

问题点

动态作用域有个难以处理之处，在改写了变量之后调用其他函数时，被调用的函数会受到影响。文字说明比较难理解，我们来看以下代码，这里函数 yobu 调用了函数 yobareru。

```perl
$x = "global";

sub yobu{
    local $x = "yobu";
    &yobareru();
}

sub yobareru{
    print "$x\n";
    # ↑输出「yobu」
}

&yobu();
```

在函 yob 中，变量值被改写为 yobu，在函数 yobareru 中显示变量 x 时，显示出来的是 yobu。就是说，函数 yob 中的变更对被调用的函数 yobareru 有影响。

引入了动态作用域后，函 yobu 中针对变量的变更对全局变量 x 没有影响，但是这个变更对函数 yobu 里进一步调用的函数 yobareru 有影响。

动态作用域中被改写的值会影响到被调用函数，因此在引用变量时它是什么样的值，不看函数调用方的代码这个是无从得知的。而逐行去看调用方代码会很麻烦。这里举的例子中代码较短，一看就能明白，但在一般情况下，函数在哪里被调用这个信息分散在代码中，是很难把握的[①]。

这个问题该如何解决呢？

① 反过来，如果要用函数调用处确定的值来改变被调用函数的行为该如何实现？参数传递是一种方法。使用参数传递后，读代码的人就知道这个变量的值是在调用处确定的。

静态作用域

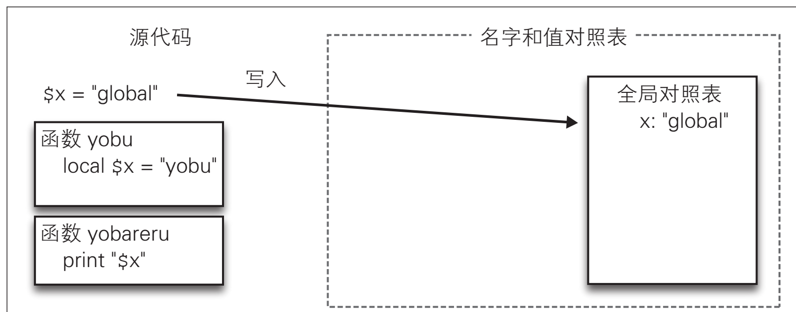
在全局作用域和动态作用域的情况下，跨越程序整体的变量内容对照表为多个函数共用，如同多人同时作业时共用白板一样。因此，函数 yobu 中改写的内容也能立刻被函数 yobareru 读取到。改写的值能被读取到是件好事，但同时也存在不希望读取到改写的值的情况。这就是动态作用域的问题点。该如何解决这一问题呢？

为函数 yobu 中临时使用的变量创建的变量内容对照表，被放在了共用的空间里，因此大家都能读取到。如果不这样做会怎样？即每调用一个函数时就创建新的对照表。就像大家每个人都使用各自桌面上的便签纸，而不是在共用的白板上做记录一样。

动态作用域中的对照表能被全部代码读取

首先，我们来看动态作用域中的代码行为。最初全局对照表中记录了“x 等于 global”（图 7.2）。

图 7.2　写入全局对照表



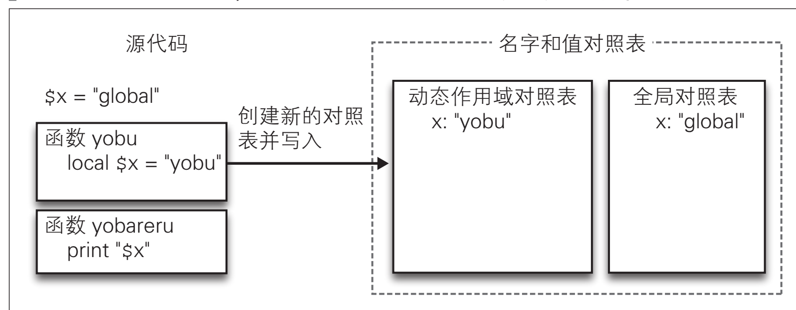
函数 yobu 中也使用了相同名字的变量 x，怎么做到不改写原来的全局变量 x 中的值呢？之前我们讨论过，把原来的值通过另一个变量名保存起来，之后再写回原来的变量名，用这种方法可以解决。通过创建新的对照表同样可以解决（图 7.3）。

使用动态作用域定义局部变量 x，要执行以下三个操作。

- 进入函数 yobu 时，准备新的对照表。

- 函数 yobu 写入变量 x 的值记录在该对照表中。
- 退出函数 yobu 时，作废该张对照表。

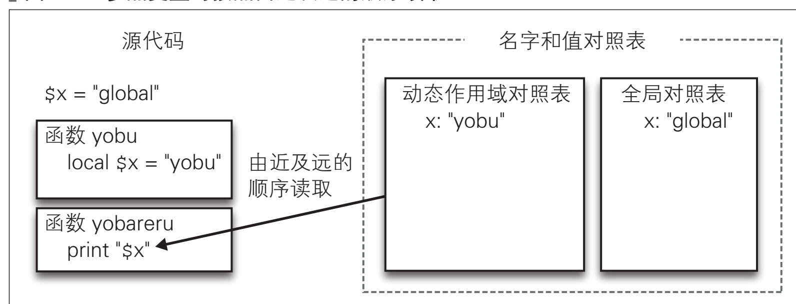
图 7.3　动态作用域下，将记录写入一张全局可见的新的对照表中



动态作用域中创建的对照表可以被全体的源代码读写，这是和静态作用域的一点很大的区别。

执行到函数 yobareru 中参照变量 x 时，还未退出函数 yobu，该张新的对照表还有效。于是程序首先读取这张新的对照表，返回其中记录的内容 yobu（图 7.4）。需要访问该张对照表中没有记录的变量时，就要翻转到下一张对照表，也就是全局对照表进行查找。

图 7.4　参照变量时按照由近及远的顺序读取



静态作用域按函数区分对照表

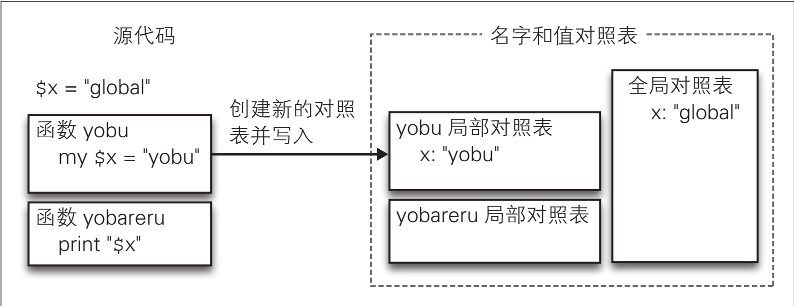
多个函数共用一张对照表，这是动态作用域的问题点。那就按函数来区分对照表吧。此时程序执行以下三个操作。

- 进入函数 yobu 时，准备函数 yobu 专用的新的对照表。

- 函数 yobu 写入变量 x 的值记录在该对照表中。
- 退出函数 yobu 时，作废该张对照表[①]。

在函数 yobu 中改写变量 x 的值时，该记录被写入函数 yobu 专用的对照表中（图 7.5）。

图 7.5　静态作用域中函数各自的对照表



随后执行到函数 yobareru 中参照变量 x 时，要访问的不是函数 yobu 专用的对照表，而是函数 yobareru 专用的对照表。这张对照表中因为没有写入函数 yobu 改写的变量值，所以全局变量的值会被读取出来（图 7.6）。

用代码来表示就是下面的样子。

```perl
Perl
$x = "global";

sub yobu{
    my $x = "yobu"; # 此处从local改为my
    &yobareru();
}

sub yobareru{
```

① 为了不招致误解需要说明一下。在从函数 yobu 中退出前，又一次进入函数 yobu 的情况也有，这时会如何？为了不使二次执行给首次执行带来影响，需要为二次调用创建第二张对照表。这里说明的三个操作是在没有递归调用的情况下退出函数 yobu 前不会再次调用函数 yobu。但一般情况下存在多次递归调用的可能性，这就需要不止一张对照表，而是要为每次函数调用配备一张对照表。

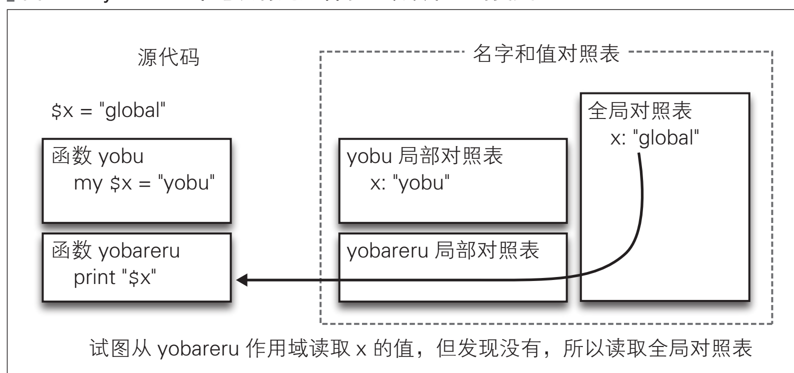
```
    print "$x\n";
    # ↑输出「global」
}

&yobu();
```

这个例子实现了一种有效作用范围，使得函数 yobu 中的变更不会影响到函数 yobu 之外的部分。它借助的就是静态作用域[①]。

这个不是很普通吗？估计有很多人会这样想。没错！现在很多语言都采用了静态作用域。

图 7.6　yobareru 本地没有定义名字 x 故读取全局变量



相对于只有全局作用域的状态，动态作用域体现了一种进步。有了动态作用域，不再需要担心万一别处也使用了名字 i，也不再需要为避免冲突而把自己使用的变量名写在稿纸上，可以安心地使用简短的变量名。然而，动态作用域也有无能为力的地方，为解决这个问题开发出了静态作用域。

现在说全局变量不好或谈避免全局污染的原因就在于此。但是，明明有减小作用范围的标准功能不用而致使问题产生，这是非常不明智的。

① 静态作用域也叫字面作用域（lexical scope），因为函数 yobu 中创建的变量的有效范围（作用域）与函数 yobu 在字面上范围是一致的。

7.3 静态作用域是完美的吗

时至今日，很多语言都选择了使用静态作用域。那么现在的静态作用域已经没有改善空间，很完美了吗？不是的。还有一些尚待解决的问题。

专栏

其他语言中的作用域

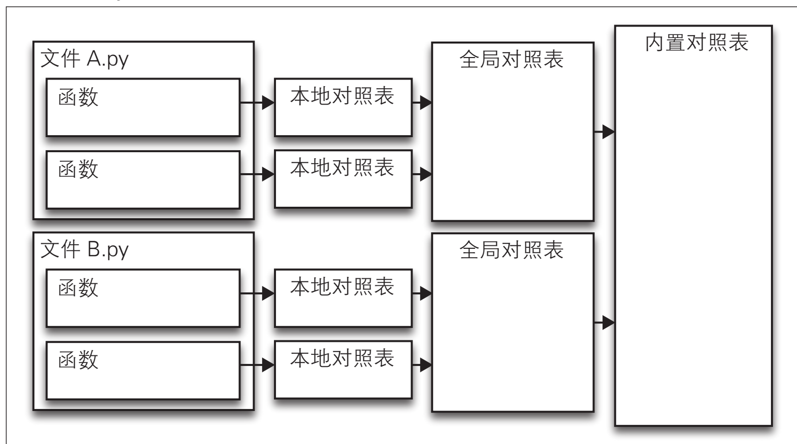
本章中的示例代码主要是用 Perl 语言写的。这是因为，Perl 语言具有对说明问题非常有利的特征，没有任何声明时使用的变量为全局变量，带 local 的变量为动态作用域（从 1991 年的 Perl 4 开始），带 my 的变量为静态作用域（从 1991 年的 Perl 5 开始）。现在执行禁止松弛代码的 use strict 模式时，用 local 声明的动态作用域变量和没有包的全局变量都会导致编译错误。

其他语言又是如何呢？ 1958 年问世的早期 LISP 语言是动态作用域。1975 年问世的作为 LISP 语言的一种的 Scheme 语言采用了静态作用域。1994 年问世的 JavaScript 语言和 Perl 语言一样，把没有任何声明的变量视为全局作用域，把用 var 声明的变量视为静态作用域。而 1991 年问世的 Python 语言和 1995 年问世的 Ruby 中，即使不带任何修饰的变量也被视为静态作用域。

在以后的程序设计中，我们应尽量避免使用全局对照表这种大家共用的空间，而去使用那些能把变更的影响范围减小的方式，写出便于理解的代码。

我们以从一开始就采用静态作用域的 Python 语言为例来说明。在 2000 年发布的 Python 2.0 中，对照表（作用域）有三个层次。范围从大到小，依次为内置的、全局的、局部的（图 7.7）。简单来讲，就是每一个程序都有一张整体对照表（内置对照表），一张文件级别的对照表（全局对照表），一张函数级别的对照表（局部对照表）。

图 7.7 Python 语言的作用域的三个层次：局部的、全局的、内置的



Python 语言的内置作用域是程序的任何地方都能参照到的对照表，比如字符串函数 str 和运行时错误 RuntimeError 这样的。因为带有包装好的函数和异常，所以被称为内置[①]。Python 语言的全局作用域是针对每个文件的对照表。有些语言也称其为文件作用域。局部作用域则是针对每个函数的对照表。

Python 语言是赋值即变量定义的语言，它没有与 JavaSript 语言的 var x 和 Perl 语言的 my $x 相当的变量声明。执行函数中的赋值，没有任何声明语句就定义了局部变量。也就是说，想要在函数中为某数值取名字，只需把数值赋值给你喜欢的名字的变量。这样做不会对函数以外的部分产生任何影响。

乍一看，这似乎是百利无害的大好事。然而人们设计的东西不管看起来是多么地正确，往往还是会有错误的地方。现在看来，这种机制至少有两个问题。

嵌套函数的问题

第一个问题就是看似嵌套结构体的作用域其实并非是嵌套结构体。

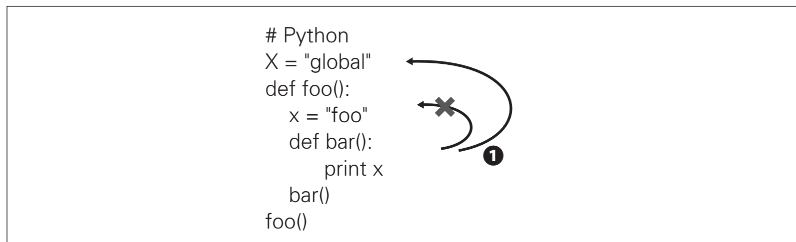
① 有些语言也可能将其称为全局的。如我们之前反复强调的一个要点，关键字用语的使用是因程序而异的。

Python 语言支持把函数作为嵌套结构体即函数嵌套，函数中允许定义新的函数。下面的代码中，函数 foo 中定义了函数 bar，来显示 x 的值。

```
Python
x = "global"
def foo():
    x = "foo"
    def bar():
        print x   ❶
    bar()
foo()
```

这句 print 语句要输出的值很直观吗？认为输出 foo 的人应该不在少数吧。然而，到 Python 2.0 情况都不是这样。这句代码输出为 global（图 7.8）。

图 7.8　容易误认为是使用代码上相近的定义



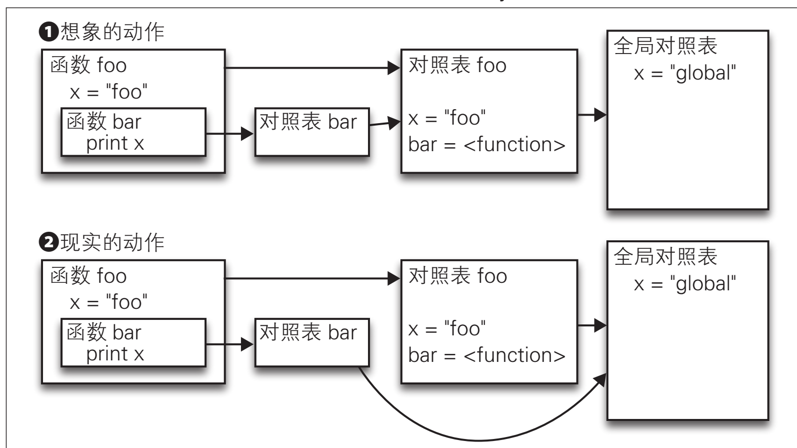
这是什么情况？从源代码表面上看，因为函数 foo 中包含函数 bar，所以很多人会以为函数 foo 的作用域中也包含函数 bar 的作用域。这就是说，bar 的作用域中找不到名字 x 时，参照相邻外部的 foo 的作用域，抱有这种想法的人很多（图 7.9 ❶）。

实际上 Python 2.0 的设计并不是这样的。函数 bar 的局部作用域中找不到名字 x 时，接下来去找的是全局作用域（图 7.9 ❷）。

这种程序设计带来了很多误解，并且有时会招致一些奇怪的解决偏方[①]。因此，认识到这是一个需要修正的问题后，2011 年发布的 Python 2.1 已经把设计修改为❶指示的逻辑。

① 利用函数参数的默认值在函数定义时确定的特点，写成 bar(x=x) 的形式把 foo 的作用域中的 x 值带入 bar 的作用域。说的是这种解决方法。

图 7.9 嵌套结构体作用域的解释：想像（❶）与 Python 2.0 中的现实（❷）



外部作用域的再绑定问题

第二个问题是无法变更嵌套作用域外部的变量。这个问题起因是 Python 语言赋值即变量定义的特点。函数中执行变量赋值语句时，这个变量就成为该函数的局部变量。赋值带来的操作是，当这个作用域中有这个名字的变量时，对该名字的变量进行再次绑定[①]，当这个作用域中没有这个名字的变量时，定义一个新的局部变量。不管哪种情况，这对于外部的作用域来说都没有影响。也就是说，无法变更此作用域外部的变量。

```
Python
def foo():
    x = "old"
    def bar():
        x = "new"
        # 打算修改外面的 x 的值
        # 却创建了一个新的本地变量
    bar()
    print x

foo() #-> old（没有能修改）
```

① 绑定，简单来讲就是将名字和值关联在一起。再绑定也就是对变量 x 关联另外的值。

Python 语言中的解决方法

对于如何解决这一问题，曾经有过非常激烈的讨论。大家提出的方案中，有像 JavaScript 语言那样在变量定义的作用域中用 var 来做声明的方法。这一方法确实能解决这个问题，但是相应的失去了与过去的代码的互换性。所以，这种方法是不可取的。

直到 2006 年 Python 3.0 中才出现了一种方法，即在函数开始时声明变量为 nonlocal 性质。

nonlocal 这个关键字的选取也是在众多备选项中挑选出来的，因为它出现的频度在过去代码中是最低的。

```
Python 3.0
def foo():
    x = "old"
    def bar():
        nonlocal x # 非本地变量声明
        x = "new" # 修改外部作用域
    bar()
    print(x)

foo() #-> new（变量修改了）
```

Ruby 语言中的解决方法

Ruby 语言也是一种不需要对变量进行声明的语言，所以它面临与 Python 同样的问题。

Ruby 语言中像函数这样发挥作用的有两种，方法和代码段（Block）。在 Ruby 1.9 版本中，方法在进行嵌套时作用域是不嵌套的（❶）[①]。

```
Ruby
def foo()
  x = "outside"
  def bar()    # 方法嵌套
    p x        #-> 出错，无法访问外部的 x（❶）
  end
```

① “PEP 3104 -- Access to Names in Outer Scopes”, http://www.python.org/dev/peps/pep-3104/

```
  bar()
end
foo()
```

另外，当方法中含有代码段时，方法的局部作用域中有的名字在代码段中被视为方法的局部变量，除此以外的被视为代码段的局部变量[1]。这也就是说，在代码段中对变量赋值时可能发生一种情况，即原本想定义一个局部变量，却因为与外部作用域中的名字重复，无意中造成了变量值的变更。在这方面需要特别注意。检查是否使用了相同名字的变量在这里并不怎么困难，因为需要检查到的范围只有一个方法。

```ruby
Ruby
def foo()
    x = "old"    # foo方法的作用域中有变量名 x
    lambda {x = "new"; y = "new"}.call    # 在方法中创建代码段
    # ↑x是方法foo的、y是方法lambda的本地变量
    p x    #-> 修改成为new
    p y    #-> 出错， y 是lambda的本地变量此处无法访问
end

foo  # 调用foo
```

7.4 小结

本章我们学习了对名字的有效作用范围进行限制的重要性。现在（2013 年）大量采用的是静态作用域。

变量在任何一种语言中都存在，但不能想当然认为它在任何语言中都是一样的，或者它从一开始就是现在这样的。事实上，语言的不同会带来各种差异，即使现在大家还在不断地进行各种讨论以寻求更好的处理方式。本章介绍了 Ruby 1.9 和 Python 3.0 中对变量作变更的例子。

虽然现在很少会使用动态作用域，但这一概念并不是完全没有用

① 像在 foo{|x, y; z| ...} 中这样，通过前置的分号来强制 z 成为局部变量。

处。与静态作用域中作用域是源代码级别上的一块完整独立的范围不同，在动态作用域中，作用域则是进入该作用域开始直至离开这一时间轴上的完整独立的范围。与此相同的特征也体现在其他好多地方。比如，在某处理进行期间，一时改变某变量的值随后将原值返回的代码编写方式就相当于创建了自己专属的动态作用域。又如，异常处理与动态作用域也很相似，函数抛出异常时的处理方式受到调用函数的 try/catch 语句的影响。

面向对象中像 private 声明这样的访问修饰符，在限制可访问范围的作用上和作用域是非常相似的。private 将可访问范围限制在类之内，而 protected 将此范围扩大到其继承类。这和函数调用处的变更会影响到调用里面的操作这一动态作用域表现是相似的，两者都具有这么一个缺点，这就是影响范围没有能限制在代码的某一个地方。

比如 Java 语言，它是静态作用域语言，它的类可以在源代码的任意处被访问。这意味着类是具有全局作用域的。但是类的名字具有层次并且只有导入后才能被使用[①]，这避免了全局变量带来的无意的名字冲突。但是不管是全局变量还是类的静态成员都可以在源代码的任意地方被变更。这提醒我们，在享受使用上的便利的同时，要谨防滥用导致的代码难以理解的情况发生[②]。

作用域是编写易于理解的代码的有力工具，很多地方都应用了这一概念。

---

① java.lang 这样的包里的类除外。

② 对于习惯了 java 语言设计模式的人，这样说或许传达更清楚些：不要把单例模式（singleton pattern）作为全局变量的替代来使用。

第8章

类型

本章我们来学习类型。

但一下子就进入类型的学习，大家也许会觉得过于抽象。

在具体学习类型的必要性前，我们先来了解一下数的表达。

以数的表达为例学习了类型的必要性后，我们了解到类型被用来达成各种不同的目的。

8.1 什么是类型

类型是什么？这很难讲。要是抽象地来解释想必效果也一般[①]。

类型是人们给数据附加的一种追加数据。计算机中保存的数据是由 on 和 off 或 0 和 1 的组合来表达的[②]。至于 on 和 off 的组合（比特列）是如何表达各种数值的，哪种比特列表示哪种值，这些只不过是人们简单的约定事项而已。同样的比特列，当其被解释为的数据的类型不同时，得到的数值是不同的。为了避免这一情况的发生，人们追加了关于数据的类型信息，这就是类型的起源。

为了把类型的意义具体化，本章我们先来学习整数和小数是如何用比特列来表现的。随后，我们会看到模样相同的比特列可能值是不同的，而如果把类型搞错了，将无法进行计算。最后，在探讨了类型作为保证不出错的方法的必要性之后，我们来谈谈类型后来是如何发展演化的。

① 数学家伯特兰·罗素（Bertrand Russell）注意到对集合的定义会产生悖论，为了回避这一问题，他提出了类型理论。如果这样抽象地介绍，大家能理解么？

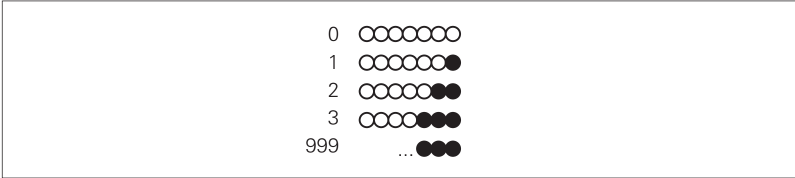
② 既有把数据当做连续量处理的模拟计算机，也有按照 10 进制计算的 ENIAC，这里不多深入介绍。

8.2 数值的 on 和 off 的表达方式

如何在电子计算机中表达数值呢？如前所述，在计算机中所有的数值都用 on 和 off 或 0 和 1 的组合来表达。为了更形象地说明，我们换个角度来思考，该如何用灯泡的点亮与熄灭来表达数值呢？

最简单的方法就是，要表达 3 这个数字时点亮三盏灯。然而在这种方式之下，需要有与希望表达的数字相当数量的灯泡，如果想表达 0 到 999 之间的数，就要准备 999 盏灯泡（图 8.1）。有没有用更少的灯泡数量来表达数字的方式呢？让我们追溯到计算机诞生以前，来看看那时数字是如何表达的。

图 8.1　点亮与欲表达数字相当数量的灯泡（黑圈表示点亮的灯泡）



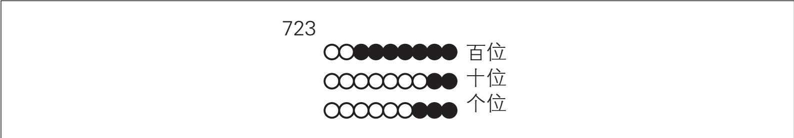
数位的发明

在计算机诞生前一千多年，人类发明了数位计算法[①]。这种方法通过在每一位上使用 0 到 9 这十个记号中的一个来表达数字。比如表达五百六十七这个数时，考虑到百位是 5，十位是 6，个位是 7，于是把 5、6、7 三个数字并列在一起。按照同样的方法，要表达 0 到 999 之间的数，只需要百位的 9 个、十位的 9 个、个位的 9 个，总共 27 盏灯泡即可

① 发明于印度，途经阿拉伯国家传到欧洲，因此后来被称为阿拉伯数字。有一本年代可考的著作是花剌子模（al-khwarizmi）于公元 825 年写的《印度数的计算法》，这是关于数位最古老的记载。值得一提的是，al-khwarizmi 这个词在拉丁语里被记为 Algoritmi，这就是现在英语中算法（Algorithm）的词源。

（图 8.2）[①]。通过数位的使用，所需的灯泡数从 999 个骤降至 27 个，那么还有没有能继续减少灯泡数量的方法呢？

图 8.2 百位、十位、个位



七段数码管显示器

在计算机诞生之前，人们早就发明了每一数位用七盏灯泡来表达数字的方法。大家肯定都在日常生活中见过它，这就是常使用于电子计算器等领域的显示数字的七段数码管显示器（图 8.3，图 8.4）[②]。

图 8.3 七段数码管显示器

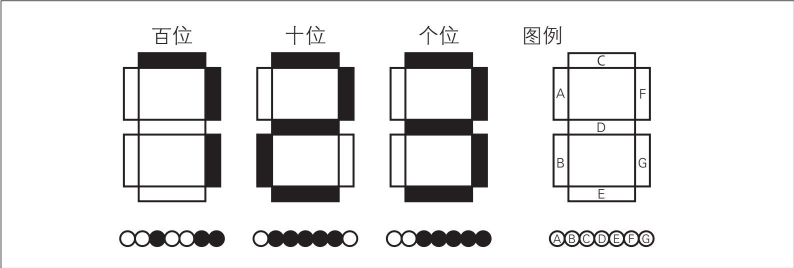
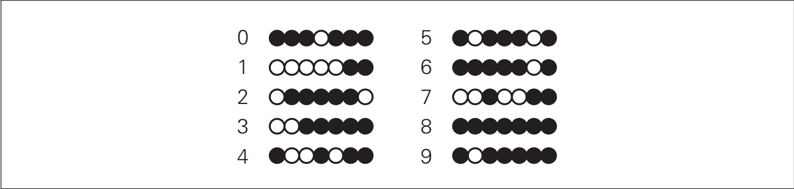


图 8.4 七段数码管显示器的数值与灯泡的对照关系



① 为什么是 9 个而不是 10 个呢？这是因为，全部灯泡熄灭能表达 0 而全部点亮能表达 9，9 盏灯泡足以表达 10 个数字了。

② 七段数码管显示器出现的年代很早，在 1908 年美国的专利申请中能发现相关记录——专利第 974、943 号：http://www.google.com/patents?vid=974943

通过使用七段数码管显示器，用七盏灯泡就可以表达一位上面的数字，表达三位的数字所需要的灯泡数量变为 21 个。这样一来又减少了 6 个。

算盘

实际上还可以继续节省灯光数量。可以做到只用 5 盏灯泡来表达一位上面的数字，这个实例就是算盘（图 8.5、图 8.6）[①]。

算盘是通过算珠的位置来表达信息的。4 个算珠用来表达 0 到 4 的数字，另外一个算珠用来表达是否需要再加上 5 这一信息。在算盘这种表达方式中，用 5 盏灯泡就可以实现一位上数字的表达，表达三位的数字所需要的灯泡数量变为 15 个。这样又进一步减少了 6 个。

图 8.5　算盘

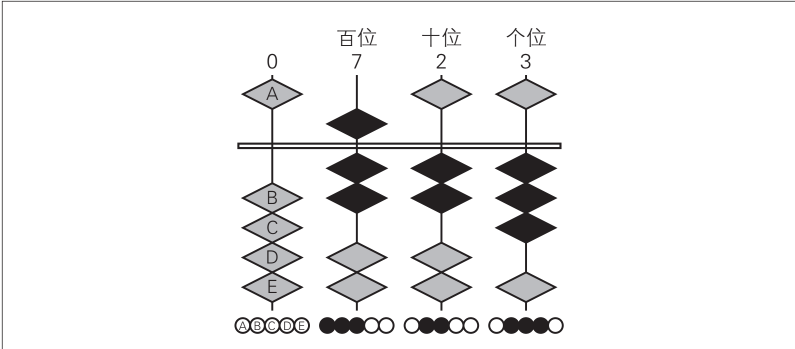


图 8.6　算盘的数值与灯泡的对照关系

0	○○○○○	5	●○○○○
1	○●○○○	6	●●○○○
2	○●●○○	7	●●●○○
3	○●●●○	8	●●●●○
4	○●●●●	9	●●●●●

① 顺便提一下，英语中的计算一词 calculate，它的词源是拉丁语中表达算盘中的算珠的词 calculus。Calcium（钙）一词的词源同样来自于此。

8.3 一个数位上需要几盏灯泡

至此，我们看到了表达数字的不同方法。它们之中有的需要的灯泡数量多，有的需要的灯泡数量少。那么所需灯泡数量最少的方法是什么呢？理论上最少又能少到什么程度呢？

一盏灯泡能表达两种不同的符号，那么两盏灯泡就能表达 4 种不同的符号。相应地，三盏就是 8 种。这还不够表达 0 到 9 之间的 10 个不同符号，因此，一个数位上三盏灯泡是不够的。有四盏灯泡的话，就可以表达 16 种不同符号了，当然也就足够表达 0 到 9 之间的 10 种符号。

事实上，早期的计算机，比如 UNIVACI 就是使用了四盏灯泡表达数值的方法。该方法被称为 excess-3（加三码）(图 8.7)。使用这种方法总共需要 12 盏灯泡就可以表达 0 到 999 之间的数[①]。

图 8.7　Excess-3 中，黑圈表示 1，白圈表示 0

0	○○●●	5	●○○○
1	○●○○	6	●○○●
2	○●○●	7	●○●○
3	○●●○	8	●○●●
4	○●●●	9	●●○○

从十进制到二进制

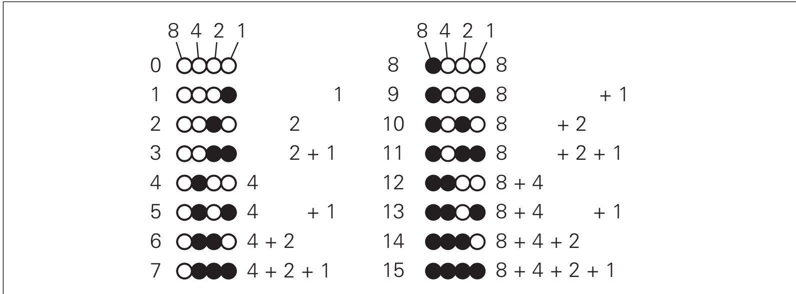
然而，四盏灯光明明足够表达 16 种符号的，而只使用了其中的 10 种符号，这总让人觉得有点浪费。有没有更加精打细算的方法呢？

这和十进制的差别在于十进制中一个数位可以表达 10 个符号。既然一盏灯泡只能表达两种不同的状态，那么进位也与之相适应，逢二进一位，这样做会怎样呢？也就是说，不再是个位、十位、百位、千

① Excess-3 中针对 0~9 中的任意一个数 x，9 减去 x 得到的数和表达 x 的四盏灯泡反转后的数是相同的。因为它具有这样的特征，用 Excess-3 来制作减法电路十分方便，这是这种符号得到普及的理由之一。

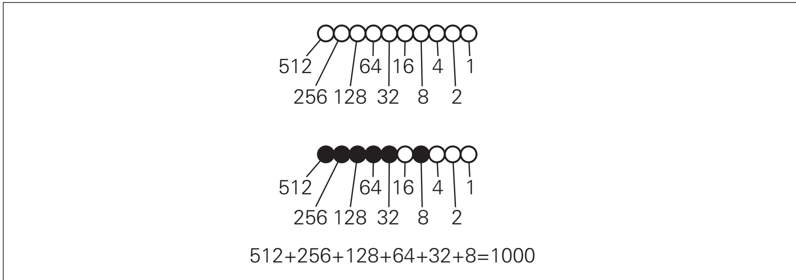
位这样地进位，而是 1 位、2 位、4 位、8 位这样地进位。这就是二进制（图 8.8）。

图 8.8 4 盏灯泡表达 16 种不同的符号



使用二进制后，10 盏灯泡就能表达 0 到 1023 之间的数字了（图 8.9）。1023 等于 1+2+4+8+16+32+64+128+256+512。上一小节讲的十进制中，要表达 0 到 999 之间的数 12 盏灯泡已经是极限了。可见使用二进制后数的表达效率能得到进一步提高[①]。

图 8.9 二进制中 10 盏灯泡表达 1000



在实际的计算机中表达整数时使用了多少灯泡呢？

1983 年，由任天堂推出的家庭计算机中使用了 8 盏灯泡[②]，能表达的整数范围是 0~255。正因如此，有些游戏中计数器到 255 时就会停止，有些游戏在有数值超过 255 时会发生程序错误。

笔者撰写这本书时是 2013 年。时至今日，PC 机使用 32 盏或 64 盏

① 十进制中的每一位大概需要 3.32 个灯泡即可，即 log(10)/log(2)。

② 严格来讲是使用了八位的 CPU。

灯泡已经成为主流[1]。32 盏能表达 0~4 294 967 295 之间的数，64 盏能表达 0~18 446 744 073 709 551 615 之间的数。

八进制与十六进制

另外，作为表达数值的方法还有八进制与十六进制。这些又是怎么回事呢？话已至此，我们一起来讲解一下八进制与十六进制。

大家平时使用的是十进制。在十进制中，每个位上使用的是 0 到 9 之间的十个符号。刚刚我们学习的二进制中，每一位上仅仅使用 0 和 1 两个符号。相比十进制，二进制在使用的符号种类更少的同时，表达相同的数字时所需要的字符数量也更多。比如十进制中的 1000 在二进制中表达就是 1111101000，这个也太长了，读起来比较困难。

把二进制中某几个字符组合在一起用一个字符来表示，使之变得更容易读，这种表达方式就是八进制或十六进制。

```
N进制中使用的字符
  二进制 0 1
  八进制 0 1 2 3 4 5 6 7
  十进制 0 1 2 3 4 5 6 7 8 9
十六进制 0 1 2 3 4 5 6 7 8 9 a b c d e f
```

八进制

例如把二进制数 1111101000，每三位三位进行切分就变成 001 111 101 000。这样切分后，每一小块有 2×2×2 总计 8 种模式。将这二进制中的三个字符通过以下表中的替换形式各自替换为一个字符后，得到的是 1750。

```
000 → 0   001 → 1   010 → 2   011 → 3
100 → 4   101 → 5   110 → 6   111 → 7
```

这里每一位可能使用八种符号，所以被称为八进制。

① 内置有 64 位 CPU 的 PC 机在一般的数码城就可以买到了，当然也有很多人还在使用老式 32 位 CPU 的 PC 机。

十六进制

再如，把同样的二进制数，按每四位四位的切分方法，得到的是 0011 1110 1000. 这样切分后，每一小块有 2×2×2 总计 16 种模式。将这二进制中的四个字符通过以下表中的替换形式各自替换为一个字符后，得到的是 3e8。

```
0000 → 0   0001 → 1   0010 → 2   0011 → 3
0100 → 4   0101 → 5   0110 → 6   0111 → 7
1000 → 8   1001 → 9   1010 → a   1011 → b
1100 → c   1101 → d   1110 → e   1111 → f
```

这里每一位可能使用十六种符号，所以被称为十六进制。习惯上，通常在八进制表示的数值前加上 0 或 0o，十六进制表示的数值前加上 0x[①]。

```
Python 3.0
>>> 0o1750
1000
>>> 0x3e8
1000
```

8.4 如何表达实数

至此，我们学习了用灯泡的点亮和熄灭来表达整数[②]的方法。接下来，我们来探讨如何表达 1.5、0.001 这样带有小数点的实数。

定点数——小数点位置确定

一种方法是确定小数点的的位置。比如，约定好把整数的小数点向

① 以 0 开始的数值，比如，在 C、Ruby、Python 等很多语言中，0100 都被当作是八进制数。但是，也有批评意见认为这样的表达方式容易被误认为是 100. Python 语言从 3.0 版本开始把 0100 作为语法错误，强制要求其表达为 0o100.

② 目前还没有接触到负数的表达方式，严格来讲这里说的是非负的整数。C 语言中，这种类型叫做无符号整数型（unsigned int）。

左移动四位，最低四位就是小数部分。这样一来，1 变成 0.0001，100 变成 0.0100 即 0.01.

这种方法有个问题，它无法表达比 0.0001 小的数，比如无法表达 0.00001。当然只要把约定改为把整数的小数点向左移动五位得到小数部分就可以，但这样针对每一个新的小数都要记一句新的约定很困难，而且还容易出错。

那该怎么办呢？

浮点数——数值本身包含小数部分何处开始的信息

把人们很难记忆的问题交给计算机去做就解决了，让数值本身包含何处开始为小数部分的信息就好了。

这是怎样一种思考方法

假如用 16 盏灯泡来表达小数，16 盏中的 10 盏可以表达 0~1023 之间的数，这是三位有效数字的信息；其余的 6 盏灯可以表达 0~63 之间的数，用来表达小数点的位置。

这种方法不仅可以表达小的数也可以表达大的数。如果把所有位数都用来表达整数，16 盏灯全部用上，最多只能表达 6 万多的数。把表达小数点位置的范围 0~63 减去 33，得到 −33~30 这一范围。把 −1 约定为小数点向左移动一位，即除以 10，+1 约定为小数点向右移动一位即乘以 10。这样一来，这种方法可以表达 1023 后面再跟 30 个零这么大的数[①]。

这就是现在一般使用的浮点数的基本思想。上一小节讲的确定小数点位置的方法被称为定点数，这个方法中的小数点是动的，所以被称为浮点数[②]。

---

① 要理解这是一个多么大的数，可以想象一下您浴缸里的水分子数量，它比这个还要大几个数量级。

② 学过 C 语言的人都知道，实数是用浮点型（float）来处理的。float 这个名称就来源于浮点小数（floating point number）。

以前关于浮点数有各种不同的约定，现在都标准化为 IEEE 754[①]。

IEEE 754 中规定的浮点数的构成

图 8.10 中的指数相当于小数点的位置，尾数相当于小数点以下有效数字的部分。

左边那盏灯（最高比特位）[②] 代表了数的符号。该位为 0 时表示正数，为 1 时表示负数 [③]。

接下来的 8 盏灯是表示位数的指数部分。指数部分作为整数理解的话可以表达 0~255 之间的数，减去 127 得到范围 –127~128。–127 和 128 分别代表了零和无限大，剩下的 –126~127 代表了小数点的位置。–126 是指小数点向左移动 126 位，127 是指小数点向右移动 127 位。

其余的 23 盏灯是尾数部分，表示了小数点以下的部分 [④]。尾数部最左边的灯泡表示 1/2（二进制中的 0.1），接下来是 1/4（二进制中的 0.01）。请看图 8.10 中的 1.75 这个数，它等于 1+1/2+1/4，用二进制来表示就是 1.11。所以，1/2 位的灯泡和 1/4 位的灯泡都点亮。指数部分为 127（要减去 127 就是范围中的 0），这表示小数点的位置移动 0 位。这两点组合起来就是 1.75。

接下来的数 3.5，用二进制来表示是 11.1。小数点向左移动一位就得到 1.11。所以它的尾数部分和 1.75 一样，1/2 位和 1/4 位点亮。指数部分变成 128（减去 127 就是范围中的 1）。3.5（二进制中的 11.1）其实就是 1.75（二进制中的 1.11）的小数点向右移动一位得到的数 [⑤]。而 7.0 则是由指数部分继续加 1 得到。

---

① 官方名称为“IEEE Standard for Floating-Point Arithmetic (ANSI/IEEE Std 754-2008)"。IEEE 754 最早制定于 1985 年，后于 2008 年进行了修订。另外，在此标准规定有 5 种标准类型，这里仅仅说明了其中的单精度二进制浮点数。

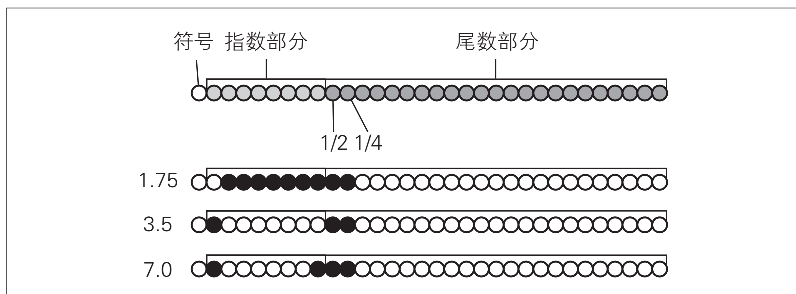
② 也可以称之为 MSB（most significant bit），最高有效位。

③ 在标准中，零区分为正的零和负的零。

④ 准确来讲，尾数是在二进制表达中为使得整数部分变成 1 而移动小数点得到的小数部分。

⑤ 在二进制中，小数点移动一位进位不是 10 倍而是 2 倍。指数部分加 1，变成 2 倍，减 1 变成 1/2。

图 8.10　IEEE 754 中单精度二进制浮点数的原理图



问题点

现今大家接触到的语言中，实数大多用浮点数 IEEE 754 表达。从实用角度来看，大部分情况下这没有任何问题。但是，这种方法要表达 3 除以 10 的答案时，十进制中可以确切表达出来的 0.3 在二进制中却变成了 0.0100110011001100110011……这样的无限循环小数，无论怎么写都有误差存在。正因为如此，会出现对 0.3 做十次加法并舍去某些位数后得到 2 这样的现象。

```
JavaScript
// 0.3做10次加法也得不到3
> 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3
2.9999999999999996

// 舍弃后变成了2
> Math.floor(0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3)
2
```

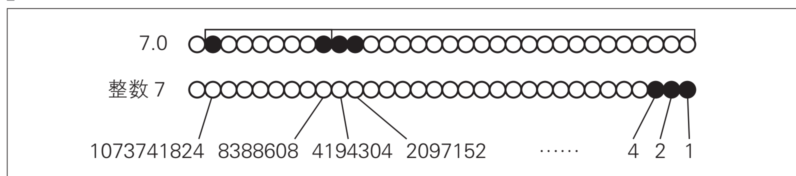
银行和外汇交易等涉及资金操作的场合尤其不欢迎这种系统行为，所以这些场合使用的是定点数或者加三码（excess-3）这样的十进制计算方式[①]。

① 一般也叫做二－十进制码，加三码（excess-3）就是其中一种。

8.5 为什么会出现类型

至此，我们学习了如何用开和关的组合（比特列）来表示整数和小数。在一般人看来，整数 7 和小数 7.0 是一样的，但在计算机看来，整数和浮点数是完全不同的（图 8.11）。

图 8.11 浮点数 7.0 和整数 7 的比特列的差异



对于计算机来说，如果仅仅给定一串比特列，它是不知道这应该解释为整数还是浮点数的。因此，需要有表示这个值为何种类型的额外的信息。这就是类型。我们来一起看一下没有类型会带来哪些麻烦。

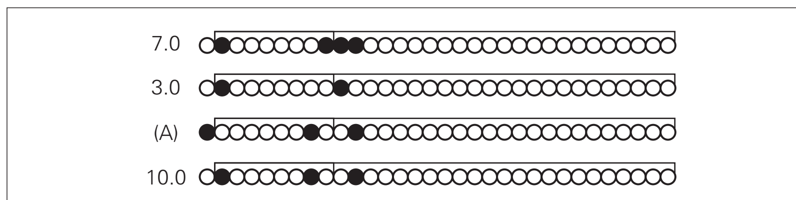
没有类型带来的麻烦

浮点数运算中 3.0+7.0 的结果是 10.0。这里用到了浮点数相加运算的命令。然而，如果忘记了 3.0 和 7.0 是浮点数而将它们作为整数做相加运算的话，结果会怎样呢？ 3.0 和 7.0 的比特列作为整数读取时将变成非常大的数值，因为左边第二个比特位在解释为整数时，已经是 1 073 741 824 这么大了[①]。相加运算的结果就更大了（图 8.12 A）[②]。

① 1 073 741 824 是由 2 自乘 30 次得到的。

② 3.0 是由 1 073 741 824 + 4194304 得到 1 077 936 128，7.0 是由 1 073 741 824 + 8 388 608 + 4 194 304+2 097 152 得到 1 088 421 888，相加的结果是 2 166 358 016。考虑上符号的话就变得更复杂了，这里只解释不带符号的整数情况。

图 8.12 7.0 和 3.0 作为整数做加法和作为浮点数做加法的结果差异



重新把这个相加的结果作为浮点数来读的话，小数点以下是 37 个 0，变成十分接近 0 的一个负数[①]。这与期待的结果大相径庭。看来，要不出现这样的错误必须多加注意。

早期的 FORTRAN 语言中的类型

在内存中记录的数值是整数还是浮点数，单靠人的记忆很难避免错误。有没有更为简易的方法呢?

一种方法是用确定的规则来表示变量名所表达的内容。比如，早期的 FORTRAN 语言使用了一系列规则，指定以 I~N 开头的变量名表示整数，除此以外的表示浮点数。

告诉处理器变量的类型

另一种更好的方法是告诉处理器某某变量是整数，让计算机而不是人去记忆这一信息。

这就是变量的类型的声明产生的原因。比如 C 语言中，声明 int x; 表示名字为 x 的变量指向的内存被解释为整数，声明 float y; 表示名字为 y 的变量指向的内存被解释为浮点数。这样通过提供关于类型的信息，处理器在进行运算时，就能自动判断该做整数相加运算还是浮点数相加运算，而不需要人们逐个去指定。

① 正确地说是 −2.93874e-38。

隐性类型转换

代码中 x+y 这样的语句，处理器是怎么执行的呢？人们可能会很直接地想到简单做个加法了事，但是对于计算机来说，整数的加法运算和浮点数的加法运算是完全不同的两件事情，不妥当地予与区分是不行的。

整数之间、浮点数之间的运算

计算机参照数据的类型来决定怎样执行。如果 x 和 y 同为整数，就做整数之间的加法运算。如果 x 和 y 同为浮点数，就做浮点数之间的加法运算。

C语言中整数 x 和整数相加

```
int x = 1, ret;
ret = x + 1024; /* ←这个加法运算 */
return ret;
```

在汇编语言中变成

```
movl     -12(%rbp), %eax
addl     $1024, %eax        # ←整数的加法运算命令
movl     %eax, -16(%rbp)
```

一边为整数一边为浮点数的运算

那如果一边是整数另一边是浮点数，该如何处理呢[1]？

在早期的 FORTRAN 语言中这将导致错误发生，因此需要程序显示地使用转换函数来指示类型转换。而在 C 语言中是自动地隐性地将整数转换为浮点数再进行运算。比如 x + 1024 中，如果 x 是浮点数，程序会将 1024 先转换为浮点数再进行加法运算。

C语言中浮点数x和整数相加

```
float x = 1.0, ret;
ret = x + 1024;       /* ←这个加法运算 */
return ret;
```

在汇编语言中变成

① 因语言不同，有时需要考虑到更多情形，比如是字符串怎么办，是时间怎么办，等等。

```
movss           -12(%rbp), %xmm0
movabsq         $1024, %rax
cvtsi2ssq       %rax, %xmm1       # ←整数到浮点数的转换命令
addss           %xmm1, %xmm0      # ←浮点数的加法运算命令
movss           %xmm0, -16(%rbp)
```

这样一来就方便很多了。这是因为，不管是整数加法运算还是浮点数加法运算，结果都不会产生太大的差异。

问题点

我们考虑一下 x / 2 这样的代码是什么意思。如果 x 为整数，除法运算后，小数点以下部分会被舍去。也就是说，在 x 为 1 时，计算结果就是 0.

如果 x 为浮点数，除法运算后变成小数点位数足够支持显示的数。也就是说，在 x 为 1 时，计算结果就是 0.5.

程序员如果事先不知道 x 的类型，见到 x / 2 这样的代码也不能判断是否会发生小数部分舍去。而让人们非记住类型不可，这和类型产生的本意是想违背的[①]。

C语言

```c
/* 仅做除法的函数 */
float divide_int(int x){
  return x / 2;
}

/* 此函数和divide_int相类似，只是 x 的类型不同 */
float divide_float(float x){
  return x / 2;
}

int main(){
  printf("%f\n", divide_int(1));     /* → 输出0.000000 */
  printf("%f\n", divide_float(1));   /* → 输出0.500000 */
}
```

① 从实用角度来看，写成 x / 2.0 的形式在 x 为整数时也将其转换为浮点数，这也是一种回避策略，但不习惯的话还是比较容易出错。

用写法来区别的语言

C 语言中采用的设计方法是由计算对象的类型来决定是否舍去小数部分。这一方法在很长时间内被很多语言使用，以至于很多程序员都非常习惯，认为理所当然。然而，这个不是恒久不变的物理法则，只不过是人们确立的设计方法而已。因此并不是所有的语言都采用这种设计。比如在 1973 年问世的 ML 语言中，整数的除法运算就表达为 x div y，而浮点数的除法运算表达为 x / y[①]。另外 1991 年问世的 Python 语言起初使用的是混杂着 C 语言风格的除法运算方式。大家从 2001 年开始讨论这种设计的恰当性，于 2008 年发布的 Python 3.0 中做了变更，把 x / y 作为与 x 和 y 类型无关不做舍去的除法运算，带舍去的除法运算用 x // y 来表示[②]。

```
Python 2.7
>>> 1 / 2
0
>>> 1 // 2
0

Python 3.0
>>> 1 / 2
0.5
>>> 1 // 2
0
```

8.6 类型的各种展开

最初为加入数值的类型信息而开始使用的类型的概念，随后被应用到越来越多的场合。这或许也是类型的概念变得难懂的原因之一吧。接

① ML 语言使用了“类型推断”理论，即程序员不做类型声明时，仅根据变量使用方式来推论其类型，它是一门专注于类型的语言。其后继语言 OCaml 中也用 x / y 和 x /. y 来区分整数的除法运算和浮点数的除法运算。

② *PEP 238 -- Changing the Division Operator*, http://www.python.org/dev/peps/pep-0238/.

下来，我们来看一下类型的不同应用方式。

用户定义型和面向对象

首先，使用语言中自带的基本数据类型通过组合定义新的类型的这一功能被发明出来，如 C 语言中的结构体。这被称为用户定义型[①]。

C语言
```
/* 整数型和字符串类型组合得到新的person类型 */
struct person {
  int age;
  char *name;
};
```

其实，不仅限于整数这样的数据，函数这样决定数据如何被处理的对象也被糅合到类型中来了。C++ 语言的设计者本贾尼・斯特劳斯特卢普把用户能自定义的类型当作构造程序的基本要素，把这种类型冠名为类。这就是第二次面向对象的发明[②]。

作为功能的类型

区别公开与非公开

后来出现了类型既是功能的观念。这种观念认为，构成结构体和类的类型不应该是全部公开而是最小限度地公开，类型是否一致这个交由编译器来检查，用类型来表达功能，与功能是否一致也是由编译器来检查。因此，只需要将与外部有交互的部分作为类型公开，而实现的细节则隐藏起来[③]。这样类型就被区分为公开部分和非公开部分了。学过 C++ 语言或 Java 语言的人应该知道 public 和 private 这样的访问控制标志。

---

① 在 C 语言之前的 COBOL 语言中，可以用基本的类型组合起来定义一种带有层次结构的记录类型。另外，PL/I 语言也有能组合基本类型并创建新的类型的语句 DEFINE STRUCTURE。结构体(structure)这个术语应该就是从那时候开始使用的。

② 关于第一次面向对象为何物，请参考第 11 章。

③ 对外部公开的部分和非公开部分要区分开来，基于这一观点的访问控制被使用于诸如 Modula-2（1978 年）的模块中和 CLU（1974 年）的抽象数据类型中。

发展为接口

将类型即功能的观念进一步延伸，就产生了不包含有具体的实现细节的类型（Java 语言中的接口等）。另外把函数是否抛出异常这一信息也当作类型的语言出现了[①]。

下面是 Java 语言中 Runnable 接口除去备注后的代码。这个接口定义了一个不带参数、不返回值（void）、带有一个名为 run 的方法的功能。

java.lang.Runnable接口

```
package java.lang;
public interface Runnable {
    public abstract void run();
}
```

我们来看这一功能在何处被使用到，比如，java.lang.Thread 中有一个构造函数 Thread (Runnable target)。它的功能就是，只要是满足不带参数、不返回值、带有一个名为 run 的方法的类，不管具体的实现细节，都可以被传递给 Thread 的构造函数。

用类型实现所有功能的时代到来了吗

类型即是功能的方法得到了越来越广泛地应用，但遗憾的是，用类型来实现所有功能的想法却还没有成功。如果它能成功，就很理想了：只要类型一致就不用关心内部的实现细节，功能与类型的不一致交由编译器来检查，编译通过意味着没有 bug。然而，仍有不少类型无法表达的信息，如输入这个数据需要多少处理时间，这个处理过程需要多少内存，线程中是否可以进行这种操作等。至今，这些问题也只能通过人为地读取文档和源代码来判断。

总称型、泛型和模板

通过将不同类型进行组合得到复杂的类型后，使用中会出现想更改其中一部分却又不想全部重新定义的再利用需求。

因此出现了构成要素部分可变的类型，即总称型。想要表现不同的

① CLU 语言和 Java 语言都会进行异常检查，详细介绍请参考第 6 章。

情况时，出现了以类型为参数创建类型的函数。C++ 语言中的模板、Java 语言中的泛型以及 Haskell 语言中的类型构造器可以说就是这种创建类型的机制[①]。

C++ 语言中

我们首先来看 C++ 语言中的模板。在用户定义型和面向对象一节中定义的 person 结构体中追加 something 这一字段，如下所示。

```cpp
#include <iostream>

template<typename T>
struct person {
  int age;
  char *name;
  T something;             ❶
};

int main(){
  person<int> x;           ❷
  x.something = 1;         ❸
  person<const char*> y;   ❹
  y.something = "hoge";    ❺

  std::cout << x.something << std::endl; // -> 1
  std::cout << y.something << std::endl; // -> hoge
}
```

这里的 something 的类型目前还没有确定。通过包括在 template<typename T>... ；范围内，它其实是声明了 T 是一个类型参数后面要代入具体的类型，而❶的写法声明了 something 的类型就是后面才确定的 T 的类型。

随后，在 main 函数中，person<int>（❷）将 int 类型代入 person 的类型参数 T 中，创建了一个新的类型。这个类型中 something 是整数型，于是就可以如❸句给它赋值整数数值了。

① 在 C++ 语言中也有以类型为参数返回函数的函数模板，这里只做简单介绍。

另外，在 T 中代入 const char* 创建新的类型后（❹），就可以如❺句给它赋值字符串（const char*）的值了。

Java 语言中

同样地，在 Java 语言中也可以实现，如下所示。

```
Java
public class GenericsTest {
    public static void main(String[] args) {
        Person<Integer> x = new Person<Integer>();        ❶
        x.something = 1;
        Person<String> y = new Person<String>();
        y.something = "hoge";
        System.out.println(x.something); // -> 1
        System.out.println(y.something); // -> hoge
    }
}

class Person<T>{
    public Integer age;
    public String name;
    public T something;
}
```

Java 语言中通过 class Person<T>，声明了这个类中的 T 是类型参数。随后与 C++ 语言中一样，通过❶句的 Person<Integer> 在类型参数中代入 Integer 类型进而创建了新的类型。

Haskell 语言中

在 Haskell 语言中可以这样实现。

```
Haskell
data Person a = MakePerson {age :: Int, name :: String, something :: a}  ❶

x :: Person Int❷
x = MakePerson {age = 31, name = "nishio", something = 1}

y :: Person String
y = MakePerson {age = 31, name = "nishio", something = "hoge"}
```

```
main = do
  print $ something x   -- -> 1
  print $ something y   -- -> "hoge"
```

通过❶中的 data Person a = ... 声明了 a 为类型参数。随后在❷中将 Int 代入类型参数中创建新的类型，声明 x 即为这一类型。

这个例子中，使用了类型参数的地方只有一个，或许它的优势体现得不太明显。然而，在 something x 中使用的函数 something 从 Person Int 中返回 Int 类型的值，在 something y 中使用的函数 something 却从 Person String 中返回 String 类型的值[①]。如果要在没有总称型功能的语言中实现，就需要针对每个现在作为参数的类型进行改写，然后逐个实现，陷入更加庞大的工作量之中。

比如 Java 语言中 list 类型 java.util.ArrayList<E> 的实现中，在追加元素的方法 boolean add (E e) 和读取元素的方法 E get (int index) 等多达 52 处使用了类型参数 E，如果一处一处改写那是相当麻烦的[②]。

动态类型

到目前为止，我们介绍的类型的机制中，处理器把变量名、保存数值的内存地址、内存里的内容的类型三者作为一个整体来看待。把类型的信息和数值看作整体的方式叫动态类型。作为其反义词，到目前为止介绍的类型机制都叫静态类型。

动态类型在 LISP 语言中已经得到应用，之后在 Smalltalk 语言中得以推广，随后因为计算机的高速化发展，其应用领域大为拓展。现在大多数的脚本语言都采用了动态类型。

比如使用了动态类型的脚本语言之一的 Python 语言中，变量声明时不需要声明类型，对同一个变量既可以赋值整数也可以赋值浮点数。

---

① 顺带一提，像这种同一个函数名下面具有多种函数实现方式的情况称为重载（多重定义，overload）。上一小节中讲到的加法运算符“+”既被用于两个整数的相加运算也被用于浮点数的相加运算，这也是重载的一种。

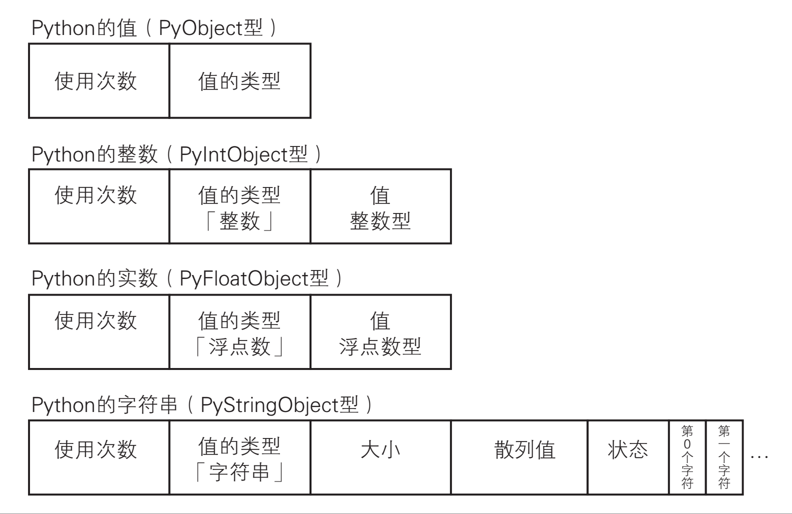
② 这里的 E 是元素（element）的首字母，它仅仅是类型参数的名称，和用 T 表示没有本质的区别。

```Python
x = 1234
x = 3.1415
```

如何实现

这是如何实现的呢？动态类型语言中之所以不需要声明类型，是因为在内存上使用了同等类型对待的设计方法[①]。比如 Python 语言中，不管是整数还是浮点数还是字符串，全部都作为 PyObject 对待，开始部分都是一样的（图 8.13）。另外在 PyObject 类型的结构中还预留了保存值的类型信息的地方。

图 8.13　Python 语言中值的开始部分结构都是一样的



※ 使用次数是指在内存管理中记录这个数值有几处被参照引用的数值（引用计数）。

※ 字符串的散列值是散列函数的计算结果（详见第 9 章），状态是表示该字符串是否记录在 Internpool 里（处理器是否把该字符串进行唯一处理的标志）。

Python 语言中的 x 变量在 C 语言中会表达为 PyObject* x。因此，要读取数值时，首先将内存上的比特列作为 PyObject 读取，便可知该数是整数、浮点数还是字符串了。再据此决定实际的数值以何种方式从内

① 这一点在其它的脚本语言中也是同样的情况，比如在 Ruby 语言中，任何数值都是 VALUE 类型的。

存中读取出来。如果是整数，表示在第三个单位里存放着整数。如果是浮点数，表示第三个单位里存放着浮点数。如果是字符串，第三个单位存放的是整数型的表示字符串长度的数，实际的字符串从第六单元开始读取。

优势与不足

使用这种数值类型处理方法，能实现历来静态类型语言不能实现的灵活处理。运行时确定类型和改变类型成为可能。然而，它也有一些不足。静态类型语言在编译时确定类型，同时编译时也检查了类型的一致性。有了这种类型检查，在实际执行前，便能发现一部分 bug。这一点动态类型语言是无法做到的。

类型推断

既不放弃编译时的类型检查，也想尽量减少麻烦的类型声明，要实现这一要求就要用到计算机自动推论确定类型的方法。

类型推断最早是 OCaml 语言和 Haskell 语言这样的 ML 语言擅长的领域，最近，在 Java VM 上运行的 Scala 语言等采用了类型推断的语言变得越来越多。

Haskell 语言和没有类型推断的 C 语言的比较

在 Haskell 语言中定义加 1 的函数 add_one 如下所示。

```
GHCi
> let add_one = \x -> x + 1
```

查询 add_one 的类型，得到的结果是取一整数返回另一整数的函数。基于 1 是一个整数而加法运算符 + 是取两个 T 类型的值返回 T 类型值的函数这两点，语言处理器得出参数和返回值都为整数的结论[①]。

① 严格来讲，在 Haskell 语言中 1 并不是整数型而是包含了 Float 等类型的型类 Num。这里的推论受“默认声明”的影响，篇幅所限，在此不多赘述。“6.3 Standard Haskell Classes” http://www.haskell.org/onlinereport/basic.html “4.3.4 Ambiguous Types, and Defaults for Overloaded Numeric Operations” http://www.haskell.org/onlinereport/decls.html#default-decls

```
GHCi
> :type add_one
add_one :: Integer -> Integer
```

来看看在 C 语言中如何定义加 1 的函数。这时候需要人为地声明参数和返回值的类型为 int。

```
C语言
int add_one(int x){
  return x + 1;
}
```

Haskell 语言的类型推断

同样是使用类型推断这一术语，在不同的语言中，如何做类型推断以及类型推断的能力如何，情况是不一样的。我们来比较一下 Haskell 语言和 Scala 语言。

首先创建一个“取出 x 返回 x”，此外什么也不做的函数 identify，再查询一下它的类型。

```
GHCi
> let identity = \x -> x
> :type identity
identity :: t -> t
```

没有做任何类型声明的情况下创建了函数 identify，查询其类型的结果是 t->t，这表示获取某种类型的参数返回相同类型参数的函数。这和预期是一致的。那么我们把 identify 作为参数来调用函数 identify 试试看。因为 identify 是返回和参数同样的类型的函数，把 identify 作为参数调用函数的话，返回也是 identify，亦即 identify 的类型该是 t->t。另外 identify identify 1 因为是 identify 1 所以最终是 1。我们来确认一下。

```
GHCi
> :type identity identity
identity identity :: t -> t
> identity identity 1
1
```

Scala 语言的类型推断

Scala 语言中类型推断的行为和 Haskell 语言是不一样的。它会首先来定义 identify 函数。如果像 Haskell 语言中那样不指明任何类型，将导致错误（❶），有必要人为指定参数的类型导入类型参数（❷）。定义好的 identify 的类型是把 T 作为类型参数接受 T 并返回 T 的函数（❸）。虽然没有人为地写明返回值的类型，但它从参数的类型中被推论出来了。

Scala的对话终端

```
scala> def identity = x => x                    ❶
<console>:7: error: missing parameter type
       def identity = x => x
                      ^


scala> def identity[T] = (x : T) => x           ❷
identity: [T]=> T => T                          ❸
```

接着来确认 identify (identify) 的类型。Scala 语言没能很好地推论出类型，显示成接受 nothing 返回 nothing 的函数（❹）。Scala 语言中的 Nothing 类型是指属于该类型的值不存在的一种特殊的类型。这意味着向该类型赋值任何参数都将导致类型错误。试着把整数的 1 赋值给它看看，的确出现了错误，指出“需要 Nothing 型的数值却被赋值 Int 型的 1 ”（❺）。

Scala的对话终端

```
scala> identity(identity)                       ❹
res0: Nothing => Nothing = <function1>
scala> identity(identity)(1)                    ❺
<console>:9: error: type mismatch;
 found    : Int(1)
 required: Nothing
              identity(identity)(1)
                                 ^
```

由此可见，同样是使用类型推断的表达方法，不同语言指示的具体内容是不一样的。刚刚展示了 Scala 语言推论失败的一个例子，即使推论成功了，在实用价值上有没有优势这个问题上，大家也是有意见分歧

的。即使承认它的优势而对类型推断的机制进行修改，在由此带来的作业代价与推论失败的代价之间做权衡之后，再决定否应该做改进和变更将是一个更加困难的问题。

强类型下是否可以做到程序没有 bug

类型推断与理论推论之间有对应关系[①]。于是有些语言发出挑战，试图通过使用比 C 语言和 Java 语言更强力的类型系统来证明程序中没有任何 bug。今后在改善类型系统的表现力和类型推断规则方面应该会开展各种研究。

然而，我们也时常听到一些关于类型的言论，有观点认为编译通过说明没有 bug 或者说可以设计出没有 bug 的程序。这些观点在多大程度上是现实的呢？这里笔者想引用日本计算机先驱人物后藤英一的文章。

> 当今，关于 SP[②] 和程序正确性的检测法方面的研究，从一开始就以没有 bug 的程序设计为目标。这个要是能百分百成功的话，到“后天”bug 就应该全部灭绝了吧。如果这个目标不能达成，大家还将继续依靠直觉和经验，使用和今天差不多的调试技术吧。也就是说，当今，比起研究如何避免 bug 产生，大家投入在研究如何及早发现 bug 并将其杀死方面的热情并不高。
>
> ——后藤英一“计算机科学的今天、明天和后天”中的程序设计语言一节[③]

※ 作者注：SP 即现在所谓的结构化程序设计的略称。

他写这篇文章的时候是 1976 年，是撰写本书的 36 年前。已经过去 36 年了，bug 并没有全部灭绝，接下来到底还要花几十年的时间才能做到呢[④]。

---

① 比如，在一个接受 X 型参数返回 Y 型返回值的函数中传递一个 X 型的数值，会得到 Y 型的返回值。这一关于类型的描述与“X 为真在如果 X 则 Y 的情况下，Y 就为真”这一逻辑的描述是相对应的，被称为 Curry-Howard 对应。

② SP 就是现在所讲的结构化编程（structured programming）的简称。

③ *Bit*，Vol.10, 1976 年 , pp.87。

④ 为了尽早发现 bug，通过频繁地进行测试达到及早发现 bug 的“测试驱动型开发”和“持续集成”等方法越来越普及，这一现象值得注意。

8.7 小结

本章我们在学习类型为何物以及类型为什么有必要之前先学习了数是如何表现的。

用 10 个印记来表现数 10 这是最原始的计数方法。人类一直在探索如何用最简洁的方式来表达数字，于是发明了进位法。并且在追求效率的过程中发明了非逢十而是逢二进位的二进制方法。二进制法在计算机的数字表现上发挥着非常重要的作用。

另外，为了表现带有小数点的数发明了定点数和浮点数。现今，虽然浮点数得到了广泛的应用，但我们也看到了它的一些不足之处。

计算机中的数值是整数、浮点数还是其他类型的数，为了在计算机中管理这一信息，于是催生了类型。起初，类型中只加入了数值的种类信息，最后又有多种多样的信息加入进来。比如，能在这个数值上施加的操作、此函数可能抛出的异常等信息都被加入到类型中来了。

现在，像静态类型和动态类型那样连内存地址和使用时间都不一样的事物也被称为类型，这使得类型这种东西变得越来越难以捉摸。什么样的信息放在什么地方，在什么样的时间被使用，从这个视角来看反而更容易理解。

专栏

先掌握概要再阅读细节

关于面对庞大信息量心力交瘁时该怎么办的问题，我们在第 6 章 6.6 节节末“学习讲求细嚼慢咽”专栏中介绍了三种方法。第二种方法就是“先掌握概要再阅读细节”。

书和文档都会有目录。浏览一遍目录便可以了解大概构造了。然后便可以开始正文的跳跃式阅读了。不要逐字逐句地读，首先看副标题和粗体字强调的内容、图表及其标题。

阅读源代码时，首先要看一眼文件夹结构和文件名。然后开始粗略地通读文件内容，对定义了的函数和类，以及经常被调用到的函数的名称要

扫一眼。

不管哪种方法，它们的共通点都是要先掌握概要再渐进式地追求细节，这是大的原则。

阅读源代码时的切入口不一样。其中一种是使用调试器中的逐步执行功能，按照执行的顺序以及调用的层次关系作为切入口去阅读的方法。它也是一样地，首先是大致掌握程序的整个处理流，然后逐渐深入到函数中的处理过程中。

用这种方法阅读，可能会有信息在脑子里过了一下又淡去了的感觉。这时就需要试一试我们的最后一种手段——从头开始逐章手抄（详见第 12 章章末专栏）了。

第9章 容器和字符串

本章我们来学习容器，它能放入多个元素。

容器有很多种类型。

为什么会有这么多类型呢?

本章会解释不同类型数据在内存中的存储方式差异，并阐述它们各自的长处和短处。

本章后半部分会介绍字符串，也就是能放入文字的容器。

文字是什么？文字的符号化又是什么？我们将边追溯历史边学习。

然后，我们会从比较分析的角度，来学习不同语言中字符串的差异。

9.1 容器种类多样

在不同的语言中，容器的名称不同，性质各异。比如，C 语言中的数组、LISP 语言中的列表、Python 语言中的元组以及 Ruby 语言中的数组。即使是名字相同，在不同语言中表达的意思也可能不一样。比如，LISP 语言和 Haskell 语言中的列表，与 Java 语言和 Python 语言中的列表在内部构造上完全不同。不同语言中名称表达的差异是导致混乱的根源[①]。因此，本章把这种存放多个元素的东西称为容器。

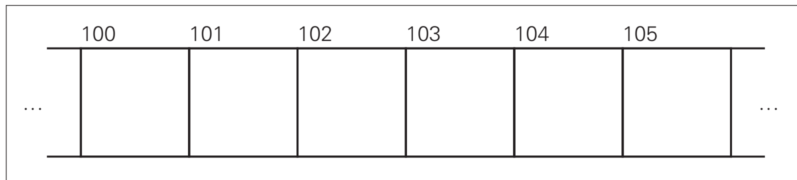
① LISP 语言中的列表指链表，有时候单说列表指的就是链表。而 Java 语言中的列表（java.util.List）则是一种有能放入多个元素的功能的接口。java.util.LinkedList<E> 就变成一种链表，这和 LISP 语言中的列表概念就比较相近了。元组也是表示能放入多个元素的东西，但 Python 语言中的元组和 Haskell 语言中的元组的共通点就很少。Python 语言中的元组是不可变更的列表，而 Haskell 语言中的列表本来就是不可变更的。Haskell 语言中的元组是可以存放不同类型数值的列表，而 Python 语言中的列表本来就可以存放不同类型的数值。Ruby 语言中的数组（array）是“数组类”，具有 C 语言数组没有的很多功能，反而更加接近 Python 语言中的列表的概念。

9.2 为什么存在不同种类的容器

为什么会有不同种类的容器呢？这是因为各种容器兼具长处和短处。

容器中的数据实际上是存放在内存中的。内存就像投币式储物柜，由固定大小的箱子按秩序排列，并编上序号（图 9.1）。容器的类型不同，内存中存储数据的方式也不同，其长处和短处正是由这些差异而来。接下来我们就来看一下存储数据方式的差异。

图 9.1　内存 固定大小的箱子按秩序排列并编好序号



数组与链表

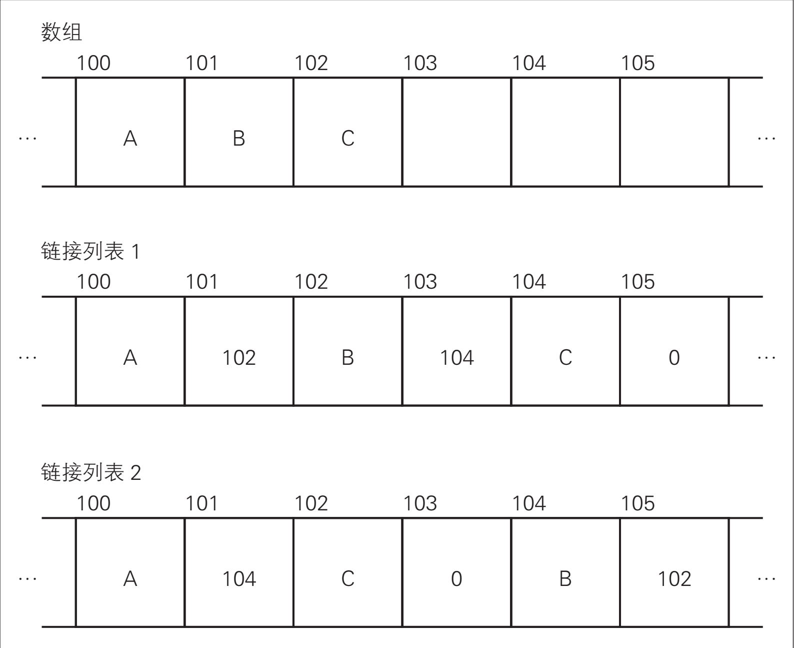
我们先来比较两种容器，一种是数组，另一种是链表[①]。

往数组和链表中都放入相同的三个数据 A、B、C，内存中会出现什么样的情形呢？其模式图如图 9.2。

在数组中，A、B、C 按顺序存放着，非常直观简单。而在链表中，存放了 A 数据的下一个箱子中还存放有表示下一个值在何处的信息。存放 C 数据的箱子的下一个箱子中存放的是 0，表示没有下一个值了。该图表现了链表中存储数据的两种不同方式，但不管何种方式，我们都可以看出 A、B、C 三个数据是按此种顺序存放进去的。

① 链表的英文是 Linked List，发明于 1955 年左右。链表常被认为是一种数据结构。其实数据结构是一个很宽泛的概念，它不仅仅指能放入多个元素的东西，还包含更广泛的含义。

图 9.2　内存中的数据存放方式

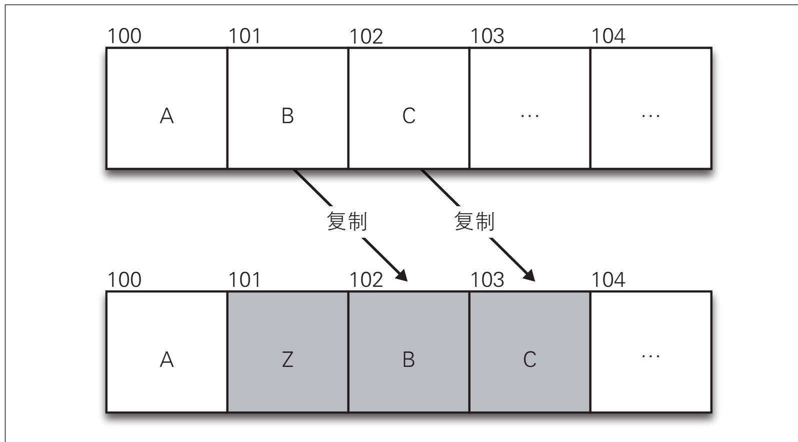


往数组中插入数值

乍一看，链表的这种数据存放方式要消耗两倍于数组的内存，有些浪费。然而，这种方式的长处也很明显。其一就是插入数值所需的时间。比如要在 A 和 B 之间插入数值 Z，该如何操作呢？

数组中的存放方式是数值按顺序排列存放的，也就是说，要先把 Z 放入 101 号箱子，再分别把 B 和 C 重新挪入 102 号和 103 号箱子中（图 9.3）。往数组中插入数值时，要求把插入此位置的所有元素都重新挪到别的位置去（复制）。

图 9.3 往数组中插入数值需要进行复制



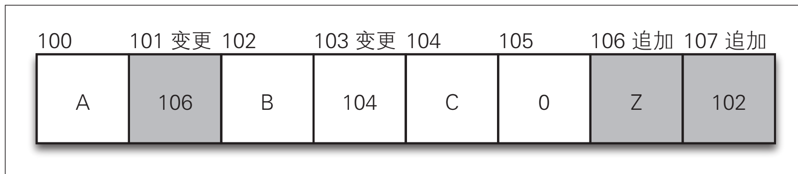
这个例子中因为数组中存放的数据量不多，只需复制两次就可以了。但如果要在一个有 10 000 个元素的数组的初始位置插入一个元素，有要进行 10 000 次复制，工作量可想而知。

往链表中插入数值

与之相对地，链表不要求在内存中按顺序存储，而是在内存中同时存放表示下一个数据存放位置的信息。

往链表中插入数据，首先要在合适的位置插入数据 Z（图 9.4）。

图 9.4 链表中插入数据只需放入两个新的箱子并修改一个箱子的内容就 OK



图中是在 106 位置插入的箱子，事实上，只要是空着的位置，在哪插入都没有关系。Z 的下一个箱子中存放了 Z 的下一个值所在的位置信息。该信息内容是 102，表示下一个值存放在 102 位置。最后，在 101 号箱子中存放的 A 的下一个值所在的位置信息被替换了，换成 A 的下一个值即 Z 所在的位置，也就是 106 号箱子。

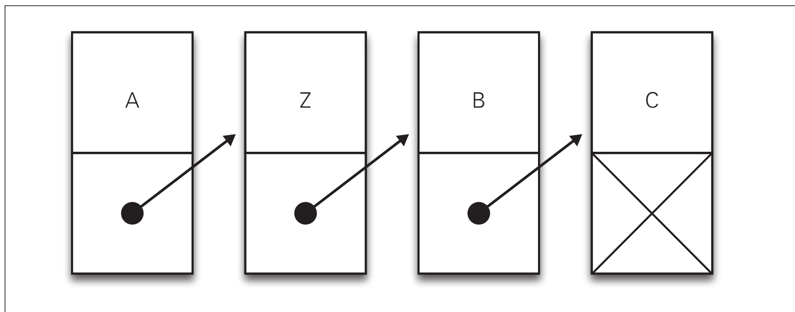
对于链表来说，只需要添加两个新的箱子并修改一个箱子的内容，便可以实现新的元素的插入。即使有 10 000 个元素，这个工作量也是不变的。

在元素较少时，使用数组还是链表的差别并不大。但是随着元素个数的增加，数组所需时间不断增加，而链表所需时间却没有变化。所以对于元素多、插入操作频繁的情况，链表是更适合的。

链表的模式图

数组在内存中的存储方式是连续的，大前提就是要有连续的内存区域。而链表却没有这一要求，它可以使用零散细分的内存区域。与数组不一样，对于链表来说，这些零散的区域在内存中所处的位置并不太重要。因此有如图 9.5 所示的一般表现形式，不明确指示出在内存的哪个位置，而只是用箭头来表示指向关系[①]。

图 9.5 链表的模式图



链表的长处与短处

本节我们用大 O 表示法来说明链表的长处与短处。

链表的长处是插入元素的计算量为 $O(1)$，而对于数组是 $O(n)$。元素的删除同样如此。链表只需要修改表示下一个元素所在位置的信息，因此计算量为 $O(1)$。而对于数组则需要把删除元素之后的所有元素都挪

① 这里的箭头表示位置信息，类似 C 语言中的指针。但不太赞成用指针这个名称的人比较多，这里姑且不使用指针这个词。

位，因此计算量为 $O(n)$。

另一方面，链表也有其短处，要获得第 n 个元素需要花费较长时间。

因为数组中的存放方式固定的，因此很容易得到第 n 个元素。比如要想知道从 100 号位置开始的数组中的第 10 个元素，在 100 上加上 10，得到的 110 号位置上读取到的数据就是它了。这时，计算量是 $O(1)$[①]。

而链表可以把各元素存放在喜欢的任何位置，因此无法使用该方法获得第 n 个元素。比如要知道从 100 号位置开始的链表的第 10 个元素，首先要读取最前面的元素，找到下一个元素所在的位置，再读取下一个元素，然后再找到其下一个元素所在的位置，如此反复操作 10 次才可以得到第 10 个元素。这时，计算量是 $O(n)$。

所以说，数组和链表都有各自的长处和短处。在实际使用中要留意自己使用的容器类型和特点，以及它是否和自己所要达到的目的相符。

专栏[②]

大 O 表示法——简洁地表达计算时间和数据量之间的关系

在此，我们先介绍一种能简洁方便地表达计算所花时间和数据量之间关系的方法。

如果是像数组中的插入一样的操作，当数据量 n 翻倍时，计算所花费的时间也翻倍，这种性质用 $O(n)$ 表示，读作 n 的数量级[③]。

而如果是像链表的插入一样的操作，即使数据量 n 翻倍，计算所花时间也不变，这种性质用 $O(1)$ 表示，读作常数的数量级[④]。

除此之外，当数据量变为 2 倍、3 倍时，计算时间增加到 4 倍、9 倍，这个用 $O(n^2)$ 表示；当数据量变成 2 倍时增加的计算时间，和数据量从 2 倍

---

① 数组的第一个元素称为第 0 个。

② 本专栏仅针对大 O 表示法进行大致说明，在此不涉及严密的定义，读者可以从其他著作中了解。如《C 语言による最新アルゴリズム事典》(中文译名：基于最新算法的 C 语言实现大全）一书中关于 $O(n)$ 有如下解释：对于常数 C（>0），如果存在数 N，使得当 $n \geqq N$ 时 $|f(n)| \leqq c|g(n)|$ 一定成立，那么当 $n \to \infty$ 时 $f(n) = O(g(n))$ 成立。

③ 也称作线性时间。

④ 也称作常数时间。

增加到 4 倍时增加的计算时间相同，这种情形用 $O(\log n)$ 表示。对于大量数据，进行 for 循环是 $O(n)$，进行二重 for 循环是 $O(n^2)$。

随着 n 的增大，大致有以下关系：

$O(1) < O(\log n) < O(n) < O(n^2)$

这种表达方式只是非常粗略地表现出函数的增加速度和方式，因此不管是 n^2+2n+1 还是 $3n^2$ 都表达为 $O(n^2)$。这样写是为了省去一些细节。

语言的差异

在 Java、Python、Ruby 等语言中都将数组作为一种最基本的容器标准[①]。比如 Python 语言中的 [1, 2, 3]，里面的元素在内存中是排列好存放的。

与此相对应，在 LISP、Scheme、Haskell 等语言中都将链表作为一种最基本的容器。LISP 语言中的 (1 2 3) 和 Haskell 语言中的 [1, 2, 3] 表达的是当前值以及下一个值存放位置信息的集合。

不言而喻，大多数语言中的容器都是在库中提供的。Python 语言中的 collection.deque 是使用链表的容器[②]，Haskell 语言中的 Data.Array 是使用了数组的容器。

9.3 字典、散列、关联数组

本节我们来看另一种诸多语言都支持的容器，它被称作字典、散列或关联数组等[③]。

---

① 数组在对象的包装下辅以一些便利的操作方法的情况很多见。

② 准确来讲，前面介绍的链表包括下一个数值的位置信息，而 Python 语言的 deque 在此基础上还包括前一个数值的位置信息，是一种双向链表。

③ 在脚本语言中常用散列一词来指代这种容器。它源自于字典实现方式之一的散列表，因此严格来讲把它和字典并列起来并不合适。为了使本章内容易于理解，暂且这样处理。

本章使用字典这个词。字典和上一节中学习的数组有什么区别呢？数组是整数和值的对应。当被问到第 10 个数值是多少时，它能告知第 10 个数值是 3。与之相对应，字典是字符串和数值的对应。当被问到 "age" 的值是多少时，它能告知 "age" 的值是 31（图 9.6）。这里的字符串被称作字典的"键"[①]。笔者认为字典这一名称很自然就能让人想起字符串和值的对应关系[②]。

图 9.6　数组与字典

数组

0	A
1	B
2	C
⋮	⋮

字典

"name"	"Taro"
"age"	31
"score"	80
⋮	⋮

我们来看一个具体的例子。有三个小孩，名字分别是 Ichiro、Hanako 和 Jiro，年龄分别为 5 岁、4 岁和 3 岁。我们来考虑一下如何在内存上记录他们名字和年龄的对应情况。实现存放多个数值的目的的方法有数组和链表这两种。同样地，实现字符串和值对应存储的也有多种实现方法。常用的方法是散列表和树。

散列表

散列表使用以字符串为参数返回整数的散列函数，实现了字符串与值的对应。存放值之前首先准备一个大的数组，然后使用散列函数（分

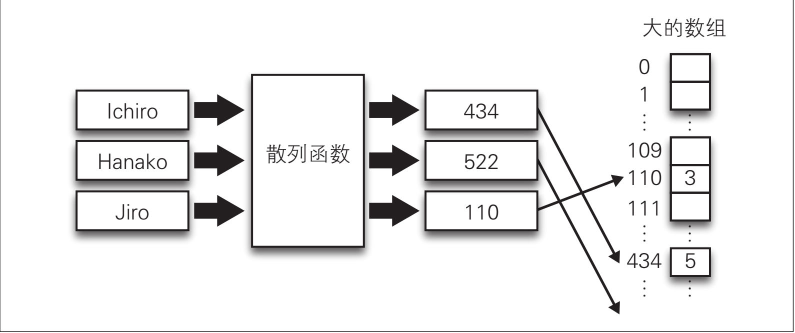
① 在实现中，字典中的键也可以是字符串以外的东西，像 Python 语言中就可以。这一点跟此处论述的主题无关，因此不多赘述。

② 敏锐的读者可能一下子想起了第 7 章中出现的名字和内容的对照表，两者颇为相似。的确如此，对照表（命名空间）和字典在功能上是等价的。"Python Reference Manual" (http://docs.python.org/release/1.6/ref/execframes.html) 中有如下论述：Namespaces are functionally equivalent to dictionaries (and often implemented as dictionaries).

散函数[①]）将字符串转换成适当分散的整数,用来决定这个值在数组中存放的位置。

首先使用散列函数把键转换为整数 n。比如 Ichiro 是 434，Hanako 是 522，Jiro 是 110. 经过这样的转换后，再把值存入数组的第 *n* 个位置中（图 9.7）[②]。

图 9.7 散列表



树

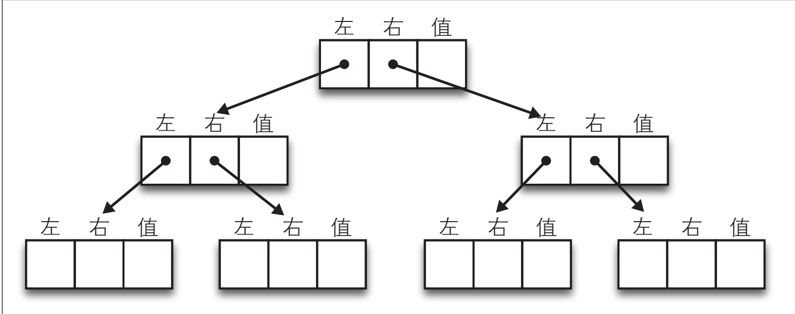
如图 9.8 所示，树是一种数据结构。链表中用箭头连接了下一个值，而树用箭头连接了右边子树和左边子树两处[③]。因为在制作图表时一般会习惯性地向下描绘，所以与其说是树不如说更像是树根。

① 分散函数这个名称最早出现于 1980 年的《信息处理手册》(日本信息处理学会编)。这个词非常直观地表达了函数的目的，但目前最为广泛的称呼还是散列函数。

② 这里仅仅介绍了概要，在实际的实现过程中会碰到几个棘手的问题。比如，不同的键传递给散列函数碰巧得到相同的内存地址该怎么办？又如，为了得到分散的整数返回值该选用怎样的散列函数？再如，事先准备好的数组多大才合适？对这些问题感兴趣的读者可以阅读一些与散列表实现相关的文献。

③ 树最上面的节点叫根，箭头的尾端叫父节点，尖端叫子节点。子节点最多两个的树被称为二叉树。一般的树也可能有三个以上的子节点。第 3 章中我们曾经讲到过结构树，本章讲的是二叉树。

图 9.8 树的结构



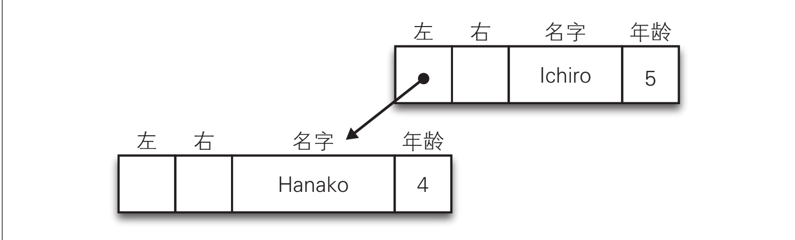
我们来看一下树的构造过程。首先把 Ichiro 作为根节点，如图 9.9。

图 9.9 以 Ichiro 为根节点



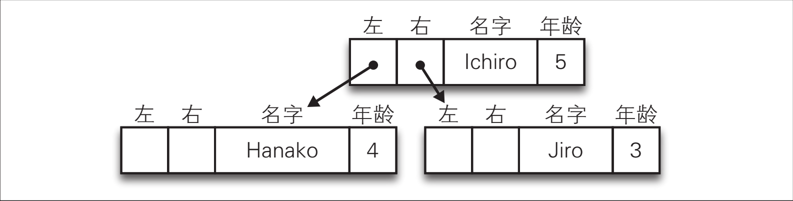
再追加 Hanako 这个元素。基本原则是键小的放在左边，键大的放在右边。把 Ichiro 和 Hanako 这两个键按照字典序比较，可知 H 在 I 的前面，因此 Hanako 要比 Ichiro 小，在其左边（图 9.10）。

图 9.10 Hanako 在 Ichiro 的左边



然后追加 Jiro 这个元素。Jiro 因为比 Ichiro 更靠后，因此在其右边（图 9.11）。

图 9.11　Jiro 在 Ichiro 的右边



然后追加 Saburo 这个元素。首先比较 Saburo 和 Ichiro，S 在 I 后面，因此往右前进。右边已经有了 Jiro。比较 Jiro 和 Saburo，S 在 J 的后面，因此放在其右边。

读取数值时，也要做同样的比较。比如要把 Jiro 这个键对应的值读取出来，首先要比较 Ichiro 和 Jiro。Jiro 在后面，因此顺着右边的箭头前进。然后比较 Jiro 和 Jiro，他们是相同的，从而得到此处记录的 3 就是和 Jiro 相对应的数值 ①。

元素的读取时间

不搞这么复杂，使用两个数组不也行吗？或者说，在数组里按键 – 值 – 键 – 值这种方式存放数据不也可以吗？抱有这种想法的人应该不在少数吧。

元素的数量较少时这样做是没有问题的。至于给定键读取对应的值时所需时间是多少，我们从大 O 表示法的角度来探讨一下。在数组里存放键和值的这种方式中，要读取特定键对应的值时，因为不知道键存放的位置，所以要从数组的开头位置按顺序往后读。或许在一开始很快就能找到，或许要到最后才能找到，平均需要进行 $n/2$ 次的检查。因此，数量级为 $O(n)$②。

① 这里仅仅介绍了概要，在实际的实现过程中会碰到几个棘手的问题。这种方法中如果没有很好地平衡树的左右两边，则可能产生性能问题。比如，对于一个对键按降序排序好了的树,它只会往左边一个方向成长。怎么保持左右两边的平衡呢？有兴趣的读者可以通过关键字平衡二叉树或是红黑树查找相关文献。

② 可能有读者会说，将数组按照键的升序先排序好，采用二分查找的办法，时间复杂度就可以降为 $O(\log n)$ 了。的确如此。但是考虑到元素添加的过程，如何才能保持数组的已排序状态呢？这又会回到平衡二叉树的话题。

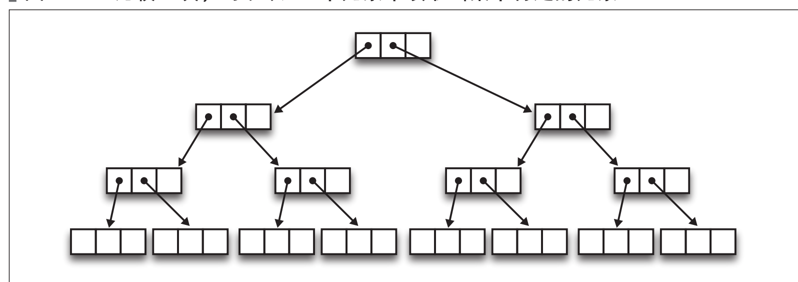
对于树

从树中读取元素是什么情况呢？在前面的例子中我们构造了一个带有 3 个元素的树。从这棵树中读取 1 个所需数据，最多需要比较两次。比如要把 Hanako 对应的值读取出来，首先把 Hanako 和根部的 Ichiro 进行比较，这是第一次。接下来，因为 Hanako 比 Ichiro 要小，于是沿着左边的箭头找到下一个，再和 Hanako 做比较，这是第二次。高度为 2 的树中最多有 3 个元素 [①]。

比较次数增加 1 次，能最多从多少个元素中读取某个元素呢？高度为 3 的树中最多能有多少个元素呢？答案是 1+3*2 = 7 个。把它想象为一个根部左右两边都是一棵高度为 2 的树的树。那这棵树的元素就是由根部的 1 个，加上高度为 2 的树中的 3 个，再加上左边高度为 2 的树中的 3 个，总共 7 个。

那么，高度为 4 的树最多又能有多少个元素呢？答案是 1+7*2 = 15 个。假设一棵树它的根部左右两边都是一棵高度为 3 的树。它的元素个数就是由根部的 1 个，加上高度为 3 的树中的 7 个，再加上左边高度为 3 的树中的 7 个，总共 15 个（图 9.12）。

图 9.12　比较 4 次，可以从 15 个元素中读取出某个特定的元素



像这样，树的高度每增加 1，元素的数量就大致变成 2 倍。反之，每当数据量翻倍时，所需的比较次数就增加 1 次。也就是说，从树中读取元素的所需时间是 $O(\log n)$ [②]。

---

① 本章讲的不是一般的树，而是子节点最多为两个的二叉树。

② 要获得 $O(\log n)$ 的性能，必须保证树的左右两边的平衡。如果向一边偏移，最坏的情况性能可能降为 $O(n)$。

对于散列表

在散列表中又是什么情况呢？要把键对应的值读取出来，首先要经过散列函数把键转换成数组中的位置信息，再把该位置上的值读取出来。这个操作与数据的量没有关系，也就是 $O(1)$ 的数量级[①]。

所以散列表所需时间数量级是最小的。正因为如此，很多语言中在制作字典时都会使用散列表。从内存的占用量来看，借助数组的方法所需内存最少，而散列表为了存放值需要很大的数组，因此内存的占用量是最大的。

没有万能的容器

也许有人要问，那到底用什么容器好呢。事实上，万能的容器是不存在的。根据容器的使用目的、使用方式和操作类型的不同，最适宜的容器类型也会相应地变化。是想要节约内存、节约计算时间，还是两样都没有必要节约。没有绝对的正确答案，而是需要根据当时的状况仔细分析，寻求最佳平衡。这是非常重要的。

比如你需要写一个针对用户操作做出响应的程序。如果在不考虑处理速度的情况下 0.01 秒可以处理完毕，那么对于这个程序还有必要做高速化处理吗？

要回答这个问题，需要重点考虑两点：什么有增加的可能性，以及其时间复杂度是什么数量级。比如开发环境中只有 10 个数据，用户实际使用的环境中有 10 万个数据。如果你写的程序的处理时间复杂度是 $O(1)$，那么即使数据有 10 万个，你也能在 0.01 秒内处理完毕。这种情况下，即使花费精力把处理速度提升两倍，时间缩短到 0.005 秒，在用户满意度提升方面带来的效益也十分有限。但是如果你写的程序的处理时间复杂度是 $O(n)$，那么数据量变为 10 万个时，处理时间就要 100 秒。耗时 100 秒用户能接受么？如果不能，那就有进行高速化处理的必要了[②]。

---

① 然而，如果散列函数的结果分散度比较差，或者为值准备的数组太小，那么不同的键的散列值碰巧落入一个内存地址的概率就会增高。这时为了不返回错误的值，就要额外花费更多的时间。

② 即使不能做高速化处理，至少也要花点工夫显示一个进度条。

9.4 什么是字符

字符串是什么？本节我们会追溯到计算机诞生以前，了解何为字符，并学习字符集和字符的编码方式。字符串一词所指的内容在不同语言中差异很大。

字符集和字符的编码方式

因国家和文化不同，提到字符，大家关联到的事物可能差异很大。在英国，人们可能会关联到 abc 等字母。在法国，人们除了会关联到字母，可能还会想到 œ、ç 这样的特殊字符。

韩国人可能把 Hangul（韩文字母）当作字符，而日本人则会把平假名、片假名和汉字当作字符。那◇是字符吗？ J 是字符吗？ ¬ 又是字符吗？

说到底，字符只不过是人们约定好的命名为字符的一系列符号的集合而已。这一符号的集合被称为字符集或字符包。字符集因国家和文化不同千差万别。

另一方面，要将字符集通过数字化的数据表现出来，就必须考虑如何对字符进行编码。编码方式也只不过是人们约定的一系列规则，有些甚至可能是随意决定的。比如，平假名中的た用 + = [①] 表示，这一规则就是随意决定的。编码方式必须为字符编码方和编码解码方共有，和没有掌握这一编码方式的人是无法进行信息交互的。

字符编码方式的发展历程，是两种观点相互角力的历史。一种观点认为应该按效率和满足个别需求的原则创造新的编码方式，与此针锋相对的观点认为过多的编码方式为信息交互带来不便，应该进行标准化。接下来我们回顾一下这段历史。

---

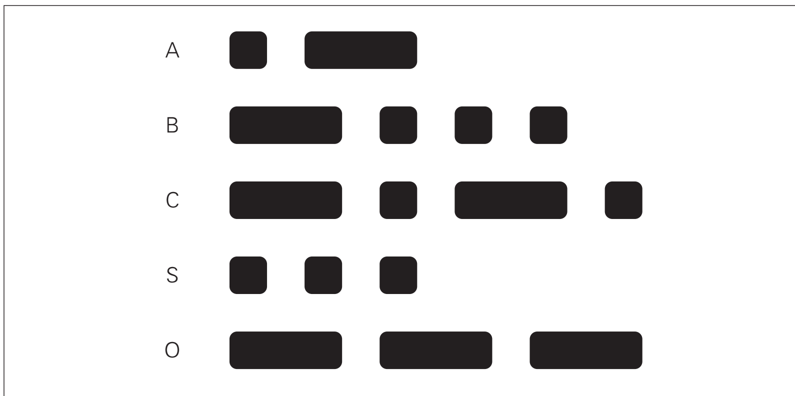
① 2002 年左右在年轻女性中流行的“少女字符”中就用 + = 来表示た。仔细看来，它和平假名中的た确实有些形似。

计算机诞生以前的编码

摩斯码

在电影等一些场合，我们经常会见到摩斯码[①]。它把用短・长・长・短・短・短・长・短・长・短[②]来表示 ABC，通过控制与无线发报机相连的按键的“通”与“断”信号进行通信（图 9.13）。

图 9.13　几个字符的摩斯码表达



摩斯码用短时间接通的短点和是其 3 倍时长的接通的长点的组合来表达字符。短点和长点之间要夹带断开状态，否则点与点之间就没有了界限，会变得无法区分和理解。点之间的断开状态时长是 1 个短点，字符间是 3 个短点，词语间则是 7 个短点。

按照这种方式来编码求救信号 SOS 时，字符串会有多长呢？ S 是短・短・短有 5 个短点时长，O 是长・长・长，有 11 个短点时长，再加上字符间的短点时长，总共有 5+3+11+3+5=27 个[③]。

① 摩斯符号（摩斯信号）这种编码方式发明于 1836 年左右，于 1865 年由 International Telegraphy Congress 纳为一种国际标准中一。

② 1836 年摩斯思考出的符号和在 1865 年标准化的符号在 C 和 F 等部分字符和数字的编码方式上有差别。

③ 在实际的电信通讯中，会将 SOS 作为特殊信息进行处理，规定在 S 和 O 之间仅保留一个短点时长。在与其他种编码方式进行效率比较时，暂不考虑电信中的特殊情况。

能用摩斯码编码的字符集有字母、数字和其他一些符号，其中字母不区分大小写。估计当时人们认为为区分大小写而增加字符的话成本较高，权衡之后决定不区分了。

博多码

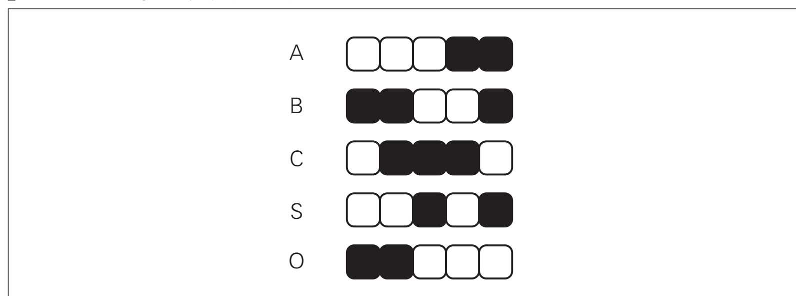
摩斯码是基于手动按压开关送信然后通过耳朵接收的设想而设计出来的一种编码方式。使用这种方式，1 秒钟能送出的量和能接收的量都很有限。随着通信需求的不断提高，迫切需要一种更为高速的通信方法。

于是后来出现了电传打字机终端。把打字机与电话线相连，通过敲击键盘输入字符，接收端通过打印机输出接收到的字符。后来，针对大量信息交互的更为高速的通信需求，出现了纸带穿孔机和读取设备相连接的方式。人们把需要传送的信息用穿孔纸带上的孔点记述下来，设备读取这些孔点后再把信息传送出去。

电传打字机终端连接的国际通信网络叫电传网，于 1931 年首次提供服务。电传网使用的编码方式是博多码，它在 1905 年被提出，在 1931 年成为了一种标准规格。

博多码的特征是，一个字符由 5 个通与断（5 比特）的组合来表现（图 9.14）。也就是说，SOS 只需要 15 个比特就能表示了。这大约是摩斯码的一半，似乎是一种效率更高的通信手段。博多码的字符集由大写字母、数字和一些其他符号组成。突出特点是它在字符集中添加了空格（不打印输出只是将光标向右移动一个字符位置的命令）、换行（光标向下一行移动的命令）、归位（光标向行首移动的命令）等控制码。

图 9.14　几个字符的博多码表达



然而，5 个比特只能表达最多 32 个字符。字母 26 个，数字 10 个，加起来就已经是 36 个了。那么如何实现 5 个比特表达 1 个字符的目标呢？

答案是使用切换。事先定义好此处是由数字模式的比特列（FIGS）起始还是字母模式的比特列（LTRS）起始，再实现两种模式的切换。这种切换叫做 shift。比如 10011 这一比特列，在字母模式下意思是 W，在数字模式下意思变成 2.

EDSAC 的字符编码

我们终于要开始展开与计算机有关的话题了。电传网出现大约 20 年后，1949 年，一种叫做 EDSAC 的计算机问世了。EDSAC 和电传网一样，用 5 个比特来表达 1 个字符，并且也用一排开有 5 个孔点的纸带进行输入。这种计算机应该比较容易生产。然后，将数字与字母通过 shift 切换输出。这一点和博多码的理念是一致的。但是，哪个字符分配到哪个比特列的分配方式和博多码是不一样的。

ASCII 时代和 EBCDIC 时代

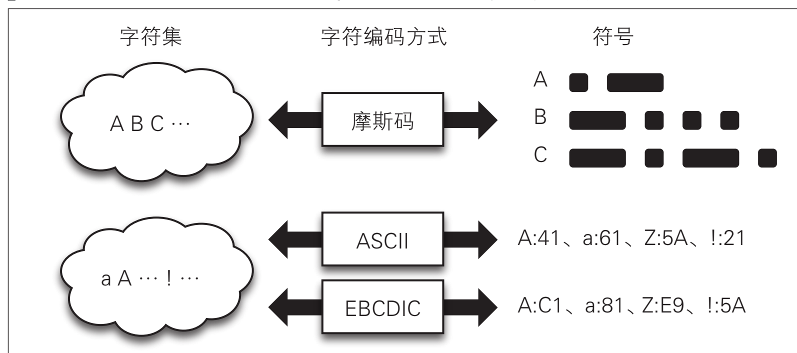
EDSAC 出现后的十年间出现了各式各样的计算机，它们使用的字符的编码方式也各不相同。这样一来，在一种计算机上读取其他种计算机输出的字符时，就需要一个个的转换，十分麻烦。为了更轻松地实现字符间的转换，人们展开了将计算机字符编码方式标准化的运动。这就是 ASCII 码，它是 American Standard Code for Information Interchange 的缩写，也就是信息交互的美国标准符号，于 1963 年被制定出来。

ASCII 中，1 个字符用 7 个比特进行编码。7 个比特可以表达 128 个字符，因此不再需要切换。ASCII 码的字符集比 EDSAC 大很多，包括大量的符号、控制码以及小写字母。

如果计算机的生产商都统一使用 ASCII 码，那么信息的交互就能变得很轻松。这应该就是当初的理想，然而现实与理想是有差距的。在 ASCII 码制定的 1963 年，当时在计算机制造市场拥有大多数份额的

IBM 公司公布了与 ASCII 码不同的 8 比特编码方式。这就是 EBCDIC[①]（图 9.15）。

图 9.15 ASCII 码和 EBCDIC 码中大小写字母和符号的编码方式



如果大家都使用同一种方法当然是更好的，但大公司往往主张大家去使用它们自家产品的方法。到现在之所以也有很多用户在使用这家公司的产品的执行方法，其中一个理由就是因为迁移成本太高。而使用其竞争厂商的产品的用户往往十分艰难，这加剧了拥有多数市场份额的公司进一步获取更多用户[②]。

最后，统一编码方式的理想未能实现，EBCDIC 自身也出现了很多种亚种。比如，比特列 01011010 对应的字符，在 HP 的 EBCDIC 中是]，在 IBM 的 EBCDIC 中是！，而在 ASCII 中则是 Z[③]。

---

① EBDIC 是 Extended Binary Coded Decimal Interchange Code 的简称，和这里的主题关系不大。字符集和 ASCII 码不完全一致，但就能显示的字符来说基本是相同的，这里做简单处理。

② 这个不仅限于发生在 IBM 身上，在操作系统市场巨头的微软公司和智能手机巨头的苹果公司身上也发生着类似的事情。

③ 谨慎起见，这里也把公司全名写出来。IBM 是 International Business Machine Corporation 的略称，HP 是 Hewlett-Packard Company（惠普公司）的略称。两者都是著名的计算机生产厂商。时至 2012 年，即使使用略称相信大家也能明白。笔者于上世纪 80 年代写的一篇文章中用了“世界上首次量产计算机的 RR 公司”，结果有不少读者没有认出来，这还曾让我烦恼过。正确答案是 Reminton Rand。它在 1955 年合并后改成了别的名称，之后又再次发生合并，演变成了今天的 Unisys Corporation。

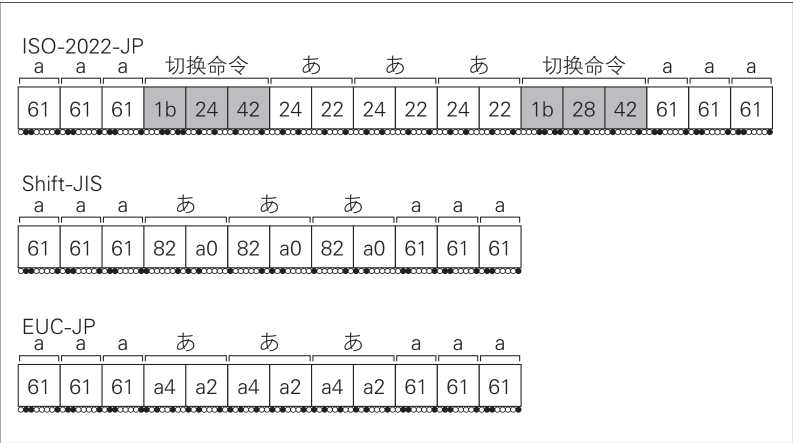
日语的编码

计算机的使用进一步普及，逐渐延伸到企业的业务领域，于是催生了新的需求。这不仅是 ASCII 中能表达的字母、数字和符号，也有对日语字符表达的需求。然而，一个字节（8 比特）最多只能表达 256 种符号，于是开始了使用多个字节表达日语中的字符。

这种编码方式也有很多种类型。这里介绍三种现在也频繁出现的类型：ISO-2022-JP[①]、Shift_JIS[②] 和 EUC-JP[③]，[④]。

图 9.16 展现的是将 aaa あああ aaa 用各种编码方式转换为字节序列的情形

图 9.16　三种编码方式中 aaa あああ aaa 的编码



---

① 也俗称 JIS 码等。

② “Windows 中使用的是 JIS”的这种说法并不正确。严格来讲，是微软公司对 Shift_JIS 进行了扩展，在 Windows 中使用了并不具有完全互换性的 Windows-31J（或称为 CP932）。

③ EUC 是 Extended UNIX Code 的略称。除 EUC-JP 外，还有针对韩语的 EUC-KR 等。

④ 本章一直使用“编码方式”这一用词。编码方式 = encode，用 en+code 来表示“使之成为编码”的意思。也有称 Shift_JIS 等编码方式为字符码的叫法。

ISO-2022-JP

在 ISO-2022-JP 编码方式中，aaa 和あああ之间有不与任何字符对应的 3 个字节，在图中用灰色显示。这是在博多码和 EDSAC 中使用的切换命令，用来切换显示字母的模式和显示平假名的模式。

在字母和符号用 1 个字节表示的模式中，24 是 $，22 是 " 。前面添上切换命令（1b 24 42，以下代码中是 \x1b$B）之后，进入平假名表达模式，24 22 作为整体，被识别为あ。

```
Python 2.7
>>> print '$"$"$"'.decode('iso-2022-jp')
$"$"$"
>>> print '\x1b$B$"$"$"'.decode('iso-2022-jp')
あああ
```

Shift_JIS

ISO-2022-JP 中单字节的字符 $ 和 双字节的字符あ的前面一个字节都是 24。Shift_JIS 编码约定，用于表现单字节的字符的字节不会被用在双字节的字符的前面一个字节上。比如，あ前面一个字节是 82，它没有用于任何一个单字节的字符。于是，即使没有切换命令也可以进行这样的判断：这个字符不是用来表达单字节字符的，而是双字节字符前面的一个字符。

EUC-JP

EUC-JP 编码约定，双字符字符中的第一个字节和第二个字节都不出现在单字节字符的表达中。

Shift_JIS 编码对程序的破坏

Shift_JIS 编码和 EUC-JP 编码的区别在于，双字节字符的第二个字节是否回避单字节字符的表达。这个区别对于程序员来讲是个很大的问题。这是因为，在 Shift_JIS 编码中，双字节字符的第二个字节有可能是程序中有特殊意义的字符。

比如，ドレミファソラシド（哆来咪发唆拉西哆）中的ソ（唆）、

表示 中的表、申し込む中的申的第二个字节和反斜杠（\）是同一个字节[1]。正因为如此，将这些字符使用在源代码中将不会得到期望的输出。

Perl
```
print("ドレミファソラシド\n");
print("表示\n");
print("申し込む\n");
```

出力
```
输出ドレミファャ宴vド
侮ｦ
垂ォ込む
```

或者将这些字符使用在字符串的最后时会导致错误发生。

```
print("図表");
```

错误输出
```
Can't find string terminator '"' anywhere before EOF at sjis2.pl line 1.
```

以上两个例子是在 Perl 程序中进行说明的。这些是在 Windows 中编写 Perl 程序时经常遇见的问题。但这一现象不是 Perl 程序中特有的。比如下面的 C++ 程序，注释符后一行代码无法执行。这是因为，注释的最后一个字符“能”字的第二个字节和反斜杠相同，因为有这个反斜杠，换行符就略过了，于是把后一行也包含在注释的内容里面了。

C++
```
#include <stdio.h>

int main(){
  printf("1\n");
  // 不好的注释：某某功能
  printf("2\n");
  printf("3\n");
}
```

① 此外，关于这个问题还可能发生在哪些字符上，请参照维基百科中 Shift_JIS 的“第二字节可能是 5C 等时的问题”部分。http://ja.wikipedia.org/wiki/Shift_JIS。

```
输出:注意 2 没有输出来
1
3
```

要解决这个问题，有人认为可以在不能正常显示的字符后面补上反斜杠的方法。也有人提议不要使用 Shift_JIS，使用 EUC-JP 或者 UTF-8 就行了。然而，这个问题原本是因为语言处理器不知道源代码是用何种编码方式编码而引起的。它不知道源代码是用 Shift_JIS 编码编写的，因此误把日语字符的第二个字节当作了 ASCII 码中的反斜杠。

魔术注释符

为了能让语言处理器正确地处理包含多字节字符的源代码，就需要告诉它源代码的编码方式。其中一个方法就是使用魔术注释符。魔术注释符最早是编辑器的一个功能。在 Emacs 和 Vim 等文本编辑器中，用特殊的记号事先写明文件的编码方式，编辑器要打开这一文件时就会以这一编码方式读取文件。

```
Emacs
# -*- coding: shift_jis -*-

Vim
# vim: set fileencoding=shift_jis :
```

语言处理器如果按这种方式去读，就能知道源代码中字符的编码了，这样一来问题就可以得到解决了。这一提案在 2001 年作为 Python 语言的扩展方案被公布出来[①]。现在 Ruby 语言、Perl 语言和 Scheme 语言的处理器 Gauche 等都采用了这一方案。

Python 语言进一步采取了更为激进的设计方法。源代码中只要是使用了 ASCII 码以外的字符，但没有使用魔术注释符时，都将导致语法错误。比如，源代码中用日语写了注释，就会带来以下错误。

---

① *PEP 0263 − Defining Python Source Code Encodings*，http://www.python.org/dev/peps/pep-0263.

Python语言中的错误消息

```
SyntaxError: Non-ASCII character '\xe6' in file tmp.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for
details
```

这段错误消息表示在文件中发现了非 ASCII 码的字符，但没有声明编码方式。

正因为此种设计，我们在编写程序时，要注意不要忘记向语言处理器声明编码方式，否则会出现一些问题。与此同时，也有用户对 Python 语言就因为仅有少量日语注释而可能导致错误这一特点表示不满。这的确是一个令人头痛的问题。

Unicode 带来了统一

至此，我们学习了表现日语字符的三种主要编码方式、它们之间的差异以及由此带来的问题。

这里我们再来回顾一下字符集和编码方式的历史。我们说过在日本国内曾经有多种字符的编码方式，听起来有些麻烦。然而把目光投向全球，这个问题就变得更加严重了。各语言表达各自的字符时都具有自己独特的字符集。编码方式必然也因国家不同而各不相同。

随着互联网的发展，各国间的数据交互越来越频繁，这个问题也越来越突显。处理希伯莱语要用 ISO-8859.8 的编码方式，处理俄语要用 KOI8，处理中文繁体字要用 BIG5，诸如此类，各国的规则必须要一一记住。那有没有更简便的方法呢？

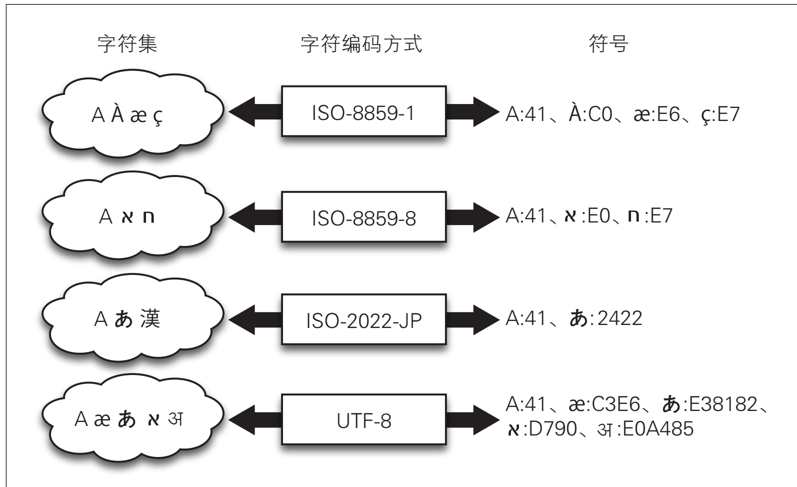
于是，我们开始了尝试设计一种能编码世界上所有字符的编码方式。1984 年国际标准化组织（ISO）开始了 Universal Character Set（UCS）的标准化作业。另外，美国施乐公司（Xerox）[①] 于 1987 年左右开始进行同样的尝试，并于 1989 年公布了初稿 Unicode Draft 1。最终，ISO 制定的规则被否决，与 Unicode 融合之后于 1993 年成为国际

① 施乐公司主要生产激光打印机，于 1970 年成立了帕罗奥多研究中心（Palo Alto），对计算机发展起到了举足轻重的作用。图形用户接口（鼠标和画面上的按钮等）和以太网（Ethernet，在 LAN 中最为广泛使用的规格）都是在这个研究中心诞生的。

标准[1]。

就这样，一个包含世界上所有字符的字符集合——Unicode 诞生了（图 9.17）[2]。

图 9.17　涵盖各国字符的统一字符集



也许有读者听说过日语字符串编码方式 UTF-8。它就是一种用来编码 Unicode 这样统一之后的字符集的编码方式[3]。

在本节的开始我们问了这样一个问题：◇是字符吗，J 是字符吗，¬ 又是字符吗？

答案是肯定的。对于这些问题，谁都可以随意去决定。但是，在这种状态下，使用时会带来诸多不便。于是大家商议出了统一的规则，这

① Unicode 是在施乐公司初稿的基础上，由相关计算机企业联名组成的非营利机构 Unicode 联盟统一制定的。

② 严格来讲，Unicode 和 ISO 制定的规格 ISO/IEC 10646（UCS: Universal Coded Character Set）不是同一种规格。其区别本书不做论述，这里不区别 UCS 和 Unicode，只进行简单说明。

③ 字符集得到了统一，但为适应不同的需求，字符的编码方式还有很多种，如 UTF-7 和 UTF-16 就是其中两种。

就是 Unicode。在 Unicode 中，这三种都规定为算是字符。

9.5 什么是字符串

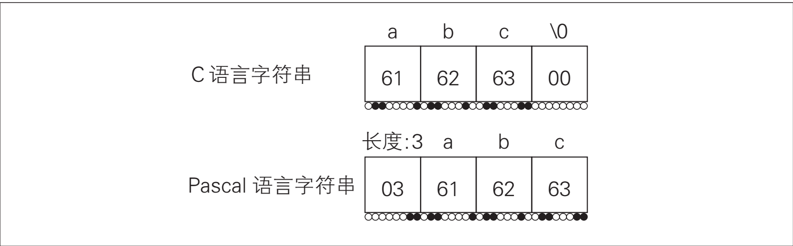
字符串就是字符并列的结果，但在不同的语言中，字符串列的表现方式各不相同。

本节我们来看一下 C、Pascal、Java、Ruby 和 Python 这几种语言中的字符串。这五种语言中，只有 C 语言中的字符串不知道自身的长度。其他语言中的字符串都携带有表现自身长度的整数。可以说 C 语言中的字符串是最为原始的字符串。

带有长度信息的 Pascal 语言字符串和不带这一信息的 C 语言字符串

C 语言和 Pascal 语言都规定 1 个字符为 8 个比特[①]。同时 Pascal 语言采用了在字符串开头放置字符串长度的规则。而 C 语言中的字符串只是拥有字符串开始后的内存空间。它不携带长度的信息，因此不知道从开始到何处为止是这个字符串（图 9.18）。

图 9.18　C 语言字符串和 Pascal 语言字符串



那么 C 语言字符串是如何表现字符串本身到何处为止呢?

① 实际上，因为 C 语言中只是规定了 char 类型为最低 8 比特的类型，IBM7.01（1952 年）和 UNIVAC 1103（1953 年）中采用了 9 比特。但是时至 21 世纪的今天，8 比特的机器占了绝大多数，这里简单化处理了。另外，char 是 character（字符）的前四个字母。

用 NUL 字符表示字符串的终止

为达到这一目的使用了一种表现字符串终止的特殊字符，这就 NUL 字符[①]。NUL 字符是一个与 0 对应的字符，在 C 语言代码中用 \0 表示。

我们来试一下这段代码。在命名为 str 的变量中放入一个在 abc 和 def 中夹着 NUL 字符的字符串。再把 str 变量放到 printf 输出，并且转递给返回字符串长度的函数 strlen。这段代码的执行结果如何呢？

C语言
```c
#include <stdio.h>
#include <string.h>

int main(){
  char str[100] = "abc\0def";
  printf("%s\n", str);
  printf("%zu\n", strlen(str));
  return 0;
}
```

结果如下所示，返回只有 abc，字符串的长度为 3.

输出
```
abc
3
```

尽管声明了 str[100]，NUL 字符后面还有 def，但这些都无影响输出结果。归根结底，C 语言字符串是把“从头开始读取，直到第一个 NUL 字符出现”的位置当作一个字符串处理。

NUL 字符导致的不便

C 语言字符串是非常原始的，因此很容易发生一些奇怪的事情。执行下面的代码得到的结果是 defabc$$，字符串的长度是 8。这是怎么造成的呢[②]？

---

① ASCII 规定将 null character 简称为 NUL，换行（line feed）简称为 LF。为了避免与 C 语言中的 NULL 指针相混淆，本书中用 NUL 字符来表述。

② 这段代码在 Mac OS X 10.7.5 上的 gcc 4.2.1 中确认过，但很可能因操作系统、编译器版本或选项的不同，执行结果也有所不同。

C语言

```
#include <stdio.h>
#include <string.h>

int main(){
  int x = 9252;
  char str[3] = "abc";
  char str2[3] = "defg";
  printf("%s\n", str2);
  printf("%zu\n", strlen(str2));
  return 0;
}
```

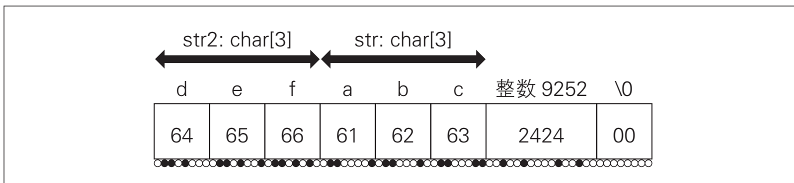
输出

```
defabc$$
8
```

原因是 str 和 str2 都声明为 char[3]，只分配了 3 字节的空间。abc 这一 3 个字符的字符串要表达它在字符 c 的地方结束的话，需要 3 个字符再加 NUL 字符总共 4 个字符的空间，但是代码中只为其分配了 3 个字节的空间。因此，abc 后面的 NUL 字符以及 def 后面的 g 和 NUL 字符都没能放入而被舍弃了。故而在显示 str2 时，首先显示 def，然后是显示与之相邻的空间里保存的 abc。

那么最后的 $$ 又是怎么回事呢？这其实是函数开始部分的 int x = 9252；语句在内存中写入的整数 9252. 9252 用 16 进制表示就是 2424，在 ASCII 码中 24 是 $。因此这个整数被解释为有两个 $ 并列的字符串的一部分。与之相邻的内存中是 00，被当作是 NUL 字符，显示到此终止（图 9.19）。然而，在某些情况下可能显示出更多的内容，并且有可能会试图读取那些禁止读取的内容，从而造成程序的异常终止。

图 9.19　运行时的内存状况



1 个字符为 16 比特的 Java 语言字符串

至此，我们学习了携带有长度信息的 Pascal 语言字符串和不带有这一信息的 C 语言的字符串。C 语言风格的字符串处理起来还是比较困难的。实际上，大多数语言都采用了 Pascal 语言风格的字符串。

Java 语言字符串也是携带有长度信息的字符串。然而 Java 语言的最大区别在于它规定 Char 类型是 16 比特[①]。C 语言或是 Pascal 语言对字符为何物的定义都不一样。C 语言中字符被定义为是一个 8 比特、0~255 范围内可以表现的 ASCII 字符或者 EBCDIC 字符，而 Java 语言中字符被定义为是一个 16bit、0~65535 范围内可表现的 Unicode 字符。

Python 3 中引入的设计变更

Python 语言既支持 Java 语言的 16 比特的 Unicode 字符串，也支持 Pascal 语言那样的 8 比特的字节串列的字符串[②]。

在 Python 2.x 版本中，源代码中有 " あ " 时，这是一个字节串列的字符串。如果源代码的编码方式为 UTF-8，这就变成一个有 ['0xe3', '0x81', '0x82'] 三个字节的串列。写成 u" あ " 时，表示这是一个 Unicode 的字符串，只有一个 Unicode 字符即 ['0x3042'].

因为同时存在两种类型的字符串，于是会有一个问题：两者混合使用的话会怎样？ Python 2.x 版本规定，在 ASCII 码环境下时字节串列被当作 ASCII 码并且可以自动转换成 Unicode。

```
Python 2.7
>>> u"hello, " + "Alice"
u'hello, Alice'
>>> u"hello, " + "太郎"
Traceback (most recent call last):
```

① 那么 16 比特无法表达的字符怎么办？这时可以使用通过两个 16bit 的值组合来表达 1 个字符的方法——代理对。它使用了 16bit 中没有用于字符表达的空间。这一原理和 EUC-JP 中通过组合 8bit 中没有用于表达字符的空间来表达 8bit 所不能表达的字符的原理是相似的。

② 严格来讲，Unicode 是 16bit 还是 32bit 是在编译时通过选项来指定的。

```
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe5 in position 0:
ordinal not in range(128)
```

※ Python 2.7 中字节串列只在 ASCII 码时才能和 Unicode 字符串结合

然而，在字符串内容不同时，这一规定有时正常有时却会导致错误。在只使用了 ASCII 字符的测试案例中可以正常运行，而在使用了 ASCII 字符以外的字符的日本却会有问题。

因此，Python 3.x 版本舍弃了 Python 2.x 版本中的兼容性，围绕字符串展开了大的变革。首先，规则发生了变化。写成 " あ " 时直接是 Unicode 字符，写成 "b" 时是字节串列，这样 Unicode 字符串就很容易书写了。" あ " 就变成和在 Java 语言中一样的了。

其次，在 Unicode 字符串和字节串列结合的时候，不管其想结合的内容如何，都将抛出类型错误。在有需要混合字符串时，规定有必要显示地使用转换代码，这避免了在不知情的情况下进行了转换而导致问题发生的被动局面[①]。

```
Python 3.0
>>> "hello, " + b"Alice"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> "hello, " + b"Alice".decode("ASCII")
'hello, Alice'
```

※ Python 3.0 中将字节串列结合到 Unicode 字符串时常常发生错误。这时需要显示地将字节串列转换成 Unicode 字符串。

※ 在转换成 Unicode 字符串时为什么要写 ASCII 呢？也许有人这样问。这里的 decode("ASCII") 是指将使用 ASCII 编码方式编码了（encode）的内容做还原处理（decode）。

Ruby 1.9 的挑战

Python、Java 等众多语言都采用了以 Unicode 为基础的字符串，而 Ruby 语言却走出了独树一帜的路线。从 Ruby 1.9 开始，字符串就是 8

① 这是我们在第 6 章中学习的“错误优先”的一个表现。

个比特，并且采用了追加编码方式信息的设计方法[①]。这种方法的优点是可以直接书写那些不包含在 Unicode 字符集中的字符。比如，在提供面向移动电话使用的互联网服务时，有需要保证各移动电话厂商设置的表情符号之间的兼容性。

在这个表情符号的问题上，谷歌公司和苹果公司采取了不同的战略。它们在 Unicode 联盟上提议将移动电话上使用的表情符号也添加到 Unicode 字符集中来。于 2010 年发布的 Unicode 6.0 中追加了数百个字符的表情符号。亦即，从 Unicode 6.0 开始，诸如“握手表情”这样的都变成字符了[②]。

9.6 小结

本章我们学习了一种能往其中放入多个元素的东西——容器。并且了解到因为在内存上存储数据的方式不一样，各种容器的性能也不同，没有一种容器在各方面都是最优的，而是优缺点兼具。多数语言都支持数组和链表两种容器。

另外，我们还学习了字符串和值的对应方式。它也会因为实现方式不同而具有各种优缺点。多数语言都支持散列表。

在本章后半部分我们一起探讨了一种可以放入任意多字符的东西——字符串。字符串也有很多种类。首先是何为字符的问题（字符集的差异），其次是如何用比特列来表示字符的问题（字符编码方式的差异），再次是如何有针对性能在内存中存储信息的问题（字符串的实现的差异）。多数语言都支持字符串，但方式绝对不是一样的。

---

① 《Rubyist Magazine - Ruby M17N の設計と実装》（中文译名：Rubyist Magazine - Ruby M17N 的设计与实现）http://jp.rubyist.net/magazine/?0025-Ruby19_m17n。

② 谷歌公司作为最大的检索引擎供应商，其提供的网络邮箱服务 Gmail 在 2013 年有同时超过 4 亿人的活跃用户。为了使得包含表情符号的邮件也能在其他终端正确显示，把表情符号添加到 Unicode 中去确实带来了很大的便利。顺便说一句，表示握手的编码是 1F359。

最后，我们讨论了试图在实现方式上解决这些差异性的问题的方案，如 Python 语言和 Ruby 语言中的实现方案。同时我们了解了试图在标准制定上解决这些问题的方案，如来自谷歌公司和苹果公司的提议。作为一名程序设计者，笔者倾向于给在实现方式上解决问题的方案给予更高的评价，但我也深知，这个地球不是单靠程序员运转的，现实中还有很多不依赖于程序实现方式的解决方案。

第10章 并行处理

本章我们学习同时进行多个处理时可能产生的问题及其规避策略。

10.1 什么是并行处理

本章我们来学习并行处理。

现在我们使用个人计算机时，可以一边听音乐，一边用文字处理软件写文章，还可以上网浏览信息。像这样，在重叠的时间段内同时进行的多个处理叫做并行处理。

在 EDSAC 等古老的计算机上，从导入程序、计算开始直到计算结束的整个过程中，除了等待不能做其他任何事情。一个程序从开始执行到其结束的期间内，计算机只能处理这一项任务。

为了实现便利的并行处理，出现了进程和线程的概念。另外，由于并行处理产生了一些新的问题，为应对这些问题又发明了锁和光纤等概念。本章我们来学习这些内容[①]。

10.2 细分后再执行

比起同一时刻只能进行一种处理，同时进行多项处理显然更加便利。然而执行处理用的线路（CPU）只有一个，怎样才能做到同时执行

① 有读者经常问起并行和并列的差别，这里做个说明。《程序设计语言：概念和结构》一书提到：程序设计语言中的并行性和硬件中的并列性是相互独立的两个概念。并列性是硬件层面的表述，比如英特尔公司于 1999 年发布的 Pentium III 中的可以同时针对四个值进行运算的 SSE 命令，以及 NVIDIA 为了记录因为 GPU 带来的高并列性的处理于 2007 年发布的 CUDA 等。而本章将要讨论的是程序设计语言领域的并行性，具体来说是进程和线程的概念。

多项处理呢[1]？答案就是在人们察觉不到的极短间隔内交替进行多项处理。尽管在某一瞬间实际只进行一项处理，但人们会觉得似乎有多项处理在同时进行。

这是并行处理中最为重要的概念。在人们看来，程序是一刻不停地在执行，但实际上它被细分成了小段来执行。

10.3 交替的两种方法

使用一个处理线路执行多项处理，就像两兄弟一起玩一台单人游戏机一样。如果能在彼此都同意的时间间隔内轮流玩，那么也就相当于两人各自在玩一台单人游戏机。“何时交替”可以分为两种情况。

协作式多任务模式——在合适的节点交替

一种方法是在合适的节点自发进行交替。利用这种方法实现的多任务（并行处理）称为协作式多任务模式。

这种方法有一个问题，有可能某个处理一直找不到合适的节点进行任务切换从而持续地进行，导致其他处理无法等到执行的机会。归根结底，采取这种方法是基于一种信任，即所有的处理都会在适当的间隔后进行交替。

再看一下那个游戏机的比喻，如果哥哥一直玩下去不给弟弟玩的话，弟弟无论等多久都玩不上了。这时弟弟估计会向妈妈告状，哥哥会被责怪吧。

Windows 3.1 和 Mac OS 9 都是协作式多任务系统。即使不是有意为之，有时也会遇到程序缺陷进入无限循环，待并行处理的程序完全没有交替，全部程序都变得没有响应了。

---

① 目前，在 PC 机的 CPU 上装载多个处理线路已经成为了主流，这称为多核。此处为了简化只讨论单核的情况。

抢占式多任务模式——一定时间后进行交替

“何时交替”这个问题的另一种解决方法是隔一定时间就进行交替。这个方法中，有一个比其他程序都具有优势的程序叫任务管理器。它在一定时间后强制中断现在正在进行的处理，以便允许其他程序执行。

还是再来看一下那个游戏机的类比，这好比妈妈每隔十五分钟命令换人玩。换成计算机，它能在人们察觉不到中断发生的间隔时间（比如 20 毫秒或 0.02 秒）实现交替。

利用这种方法实现的多任务称为抢占式多任务模式。所谓抢占就是指能够终止其他人的行为，这种方式和协作式多任务模式不同，它的显著特征在于，不需要被终止程序的协助就可以单方面强制终止它[①]。

Windows 95、Mac OS 以后的版本以及 Unix、Linux 等操作系统都是使用这种方法实现的多个程序的并行处理。

10.4 如何避免竞态条件

对程序使用者来说，抢占式多任务模式十分方便，但对于程序设计者来讲会出现其他问题。在不知道何时被喝令终止并交替的前提下，要编写一个能稳妥执行的程序是非常困难的。

我们来看下面这段反映银行交易的伪代码，看看有可能发生哪些问题。

```
如果存款余额高于10 000元 {
    存款余额减去10 000元、
    取出10 000元的钞票
}
```

单看这段程序似乎没有任何问题。问题会发生在当存款余额这个变

① preemptive 和 preemption 是军事和法律用语，字典中的解释比较难懂。牛津高阶英语词典中的解释是 done to stop somebody taking action。

量被共有，并且这个程序在多处同时执行时。在检查了“如果存款余额高于 10 000 元”之后，有可能不凑巧刚好又交替到别的程序的执行上。这时会出现以下情况：存款余额只有 15 000 元时却取出来 20 000 元钞票的严重问题。

```
（假设存款余额有15 000元。首先程序A执行）
A：存款余额高于10 000元吗？→Yes
（这里又交替到程序B的执行上）
B：存款余额高于10 000元吗？→ Yes
B：存款余额减去10 000元、取出10 000元的钞票
（这里存款余额变为5，000元。然后再交替回程序A）
A：存款余额减去10 000元、取出10 000元的钞票
（存款余额仅有5000元却取出了1万元！ ）
```

这种局面被称为竞态条件（race condition），或者说这个程序是非线程安全的。

竞态条件成立的三个条件

竞态条件在什么时候成立呢？并行执行的两个处理之间出现竞态条件必须同时满足以下三个条件。

- ⓐ 两个处理共享变量
- ⓑ 至少一个处理会对变量进行修改
- ⓒ 一个处理未完成之前另一个处理有可能介入进来

反之，只要三个条件中有一个不具备，就可以编写适于并行处理的安全的程序[①]。

为了说明ⓐ和ⓑ，我们考虑一下两个人看一台电视的场景。如果有一方随意地换台，另一方就有可能抱怨。这相当于满足了“ⓐ变量被共享”和“ⓑ有一方可能修改”的条件。如果另外买一台电视一人一台的话，无论什么时候换台都不至于招致另一方的抱怨，因为没有了ⓐ中共

① 这三个条件是参考了 *Java Concurrency in Practice* 一书（Brain Goetz，Joshua Bloch，Doug Lea 著，中文版为《Java 并发编程实战》，机械工业出版社 2012 年 2 月出版，童云兰译）关于修复问题程序的三种方法的表述总结出的。

享的情况。或者，不去换台两个人一直看，也不会引起任何问题，因为没有了ⓑ中一方修改的可能。

没有共享——进程和 actor 模型

如果最初就没有共享任何数据，条件ⓐ就不可能发生，也就没有必要在意竞态条件了。

在进程中没有内存共享

相信很多人都知道 UNIX 将执行的程序叫做进程（process）。不同的进程不会共享内存，所以在多个程序之间不会在内存上出现竞态条件。只需要注意与数据库连接或文件读写时共享数据的情形就够了。

但是进程这个词直译就是“处理”，是一个非常通用的概念。它在 UNIX 诞生以前就已经被使用。当时的进程和现在 UNIX 中的进程不一样，它有时会共享内存，有时会进行排他控制，更像是现在我们所称的线程之类。

1969 年问世的操作系统 Multics 中，进程也是共享内存的[①]。但是 Multics 项目加入了太多的功能而变得过度复杂，因此后来大家又发起了简化运动。最终，在 1970 年操作系统 UNICS 被设计出来，它就是后来的 UNIX。

UNIX 采用了一种机制，它能保证每个进程所需要的内存空间，不同进程之间不会共享内存[②]。

没有共享的方式成功了吗

UNIX 的机制是一个进程中可以并行执行的处理只有一个。也就是说，如果要想多个处理并行执行就需要启动多个进程。不同的进程不共享内存，即并行执行的处理之间不共享内存。那么这种方式后来成功了么?

没有。在 UNIX 发布大约 10 年后，人们设计出了“轻量级进程”。

---

① Multics 是基于 PL/I 和汇编语言编写的。1964 年出现的 PL/I 程序设计语言是第一个引入了类似于今天的线程概念的编程语言。

② 想了解更多细节的读者可以搜索关键字 Virtual Address Space。

它是一种共享内存、具有 UNIX 出现以前风格的进程。后来，这个被称为线程。

笔者执笔本书时 UNIX 已走过了近 40 年。在这 40 年的时间里，没有共享的方式中始终有解决不了的问题。即使现在使用线程，对于共享内存的处理仍然让人头疼，但没办法，还得继续编写程序。

actor 模型

在不共享内存的设计方针下，还有一个流派——actor 模型。它发布于 1973 年，是为实现并行处理而出现的一种模型。

它认为可以通过不共享内存而是传递消息的方法[①]来在并行处理时进行信息交互。

我们以行政文员、资料和公文格为例来说明。甲打开桌上的资料进行处理时，如果乙走过来希望甲处理其他资料，这就影响了甲正在进行中的工作，这就是共享内存的问题所在。一方面，在甲的工作告一段落之前，即使乙在旁边一直等候也是浪费时间。这就是后面要讲到的死锁的问题。如果不这样，乙在往甲的公文格中放入新的资料后马上回去处理自己的工作，这就变成 actor 模型。

这种模型中处理是非同步的。乙不知道甲何时会处理完公文格中的资料。不管何时处理完，如果在资料中写明“处理完毕请送回乙处”等信息，一旦乙在自己的公文格中看到了甲的回复，也就知道了这些资料已经处理完毕。

这种机制适宜的处理场景中，必须提一下消息交互的场景。如 Twitter 和 Facebook 这样面向大量用户的信息交互服务，应该有很多适合 actor 模型发挥作用的处理。这些处理在实现中使用了那些采用了 Erlang[②]、Scala 等 actor 模型的语言。

---

① 同期于 1971 年诞生的 Smalltalk-71 是一种通过发送消息实现信息交互的语言。*THE EARLY HISTORY OF SMALLTALK*, http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html。

② Erlang 语言在 1987 年问世，并于 1998 年成为开源的程序设计语言。2007 年，因为 Twitter 等应用的出现备受关注，是一种能让消息发送和接收变得更轻松的设计语言。

不修改——const、val、Immutable

有一种方法是通过规避条件❺，即使共享内存，只要不作修改也不会有任何问题。

Haskell 语言就是一种大力提倡这种方针的语言，它的所有变量都不可修改。

但是更多的语言采用了更加现实的折衷策略——使一部分变更无法作修改。比如在 C++ 语言中，使用 const 声明变量时，这个变量就是无法修改的。再如在 Scala 语言中，有 var 和 val 两种声明变量的方法，val 声明的变量就无法作修改。

Java 语言经常使用到 Mark Grand 提出的设计模式[①]之一的 Immutable 模式。这种模式下，类中定义了 private 字段，同时定义了读取这些字段的 getter 方法，但不定义对这些字段作修改的 setter 方法。因为没有准备用于修改的方法，所以实现了只能读取但不能改写的效果。

不介入

如果要消除竞态条件成立的第❻个条件——一个处理未完成之前另一个处理有可能介入进来，这时该如何做呢？在处理期间如何杜绝别的作业介入进来？

线程的协调——fibre、coroutine、green thread

其中有一类方法，如 fibre、coroutine、green thread 处理介入的原因是抢占式线程，那么使用协调模式的线程就可以解决了。Ruby 语言中的 Fibre 类，以及 Python 语言和 JavaScript 语言中的 generator 都是这种情况[②]。

毫无疑问，由于是协作式多任务模式，如果有某个线程独占 CPU，

---

① 想了解更多细节的读者可以参照 *Patterns in Java: a catalog of reusable design patterns illustrated with UML*（Mark Grand 著）一书。——译者注。

② Ruby 语言中的 Fibre 是在 1.9 版本中增加的，而 JavaScript 语言中的 generator 是在 1.7 版本中增加的。Python 语言和 JavaScript 语言都在 generator 中使用了 yield 一词，但 Ruby 语言由于历史原因将 yield 用于了别的目的，所以具有不一样的含义。此处不应该混淆。

其他处理就只能停止。说到底，这种方法的前提是各个线程能保证合理的执行时间在合适的时候做出让步。

表示不便介入的标志——锁、mutex、semaphore

另有一类方法，共享一种表示不便介入的标志。比如做一种约定，如果某内存上的值不为 0 时，这意味着其他线程如果介入将带来麻烦。那么各个线程在执行那些介入后可能带来问题的处理时，事先会去检查这块内存上的值。如果为 0 则继续执行，如果不为 0 则等待其变回 0 后再做处理。

这和试衣间中的门帘或单人浴室包间的状态牌类似。门帘关闭时表示这时试衣间正被占用，现在进去的话不方便。想使用试衣间的人只能在外面一直等到门帘打开为止。

这类方法有多种称法，如锁、mutex 和 semaphore，但它们的核心概念和浴室的状态牌是一样的[①]。锁这个名字很容易让人误解为只要上了锁其他人就进不来了，然而实际上它只是一个表示“使用中”的状态牌。如果有线程不去检查状态牌的状态，那它也就变得没有意义了。

这一机制是艾兹格·迪科斯彻（Edsger Wybe Dijkstra）于 1965 年发明的。1974 年霍尔（Hoare）发明了更加方便的改良版本，即 Concurrent Pascal 中采用的 monitor 的概念。1974 年时 C 语言已经问世 3 年了，直到 20 年后问世的 Java 语言采用了 monitor 的概念，它才得以广泛使用。

在进入之前先检查是否挂有“使用中”的状态牌，如果有则等待，如果没有挂则挂上“使用中”的状态牌再进入。要实现这一系列约定的动作是件比较麻烦的事情。比如使用 if 语句时，在“做值的检查”和“判断为 0 则改为 1”时，有可能有其他处理介入进来。这样一来检查就毫无意义了。为了不让其他处理在中间介入进来，就有必要使用一种能将值的检查和修改同时执行的命令。因为 Java 语言处理器实现了这一功能，所以 Java 语言用户无需烦恼，只要直接使用 synchronized lock 就可以轻松地使用锁的功能。

① 本书统称为锁。

10.5 锁的问题及对策

锁的问题

即便是使用上变得如此简便的锁，也还面临一些难题。

陷入死锁

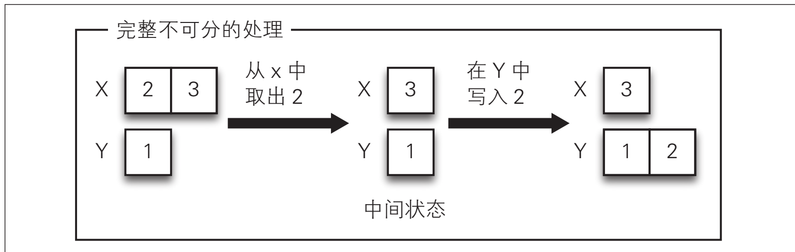
假设有 A 和 B 两个作业，它们都能修改 X 和 Y 两个变量。A 按照先锁住 X 再锁住 Y 的顺序上锁，B 按照先锁住 Y 再锁住 X 的顺序上锁，在某些时间点有可能会产生问题。比如在 A 锁住了 X 的同时 B 又锁住了 Y，双方都会等待对方释放解锁。

这一现象叫做死锁。为了避免这一问题，程序员就需要在程序的整体上注意上锁的顺序，不仅要把握应该对什么上锁，还要把握好按什么顺序去上锁。

无法组合

另外，锁还有无法组合这一个问题。比如要从列表 X 中删除第一个值然后追加到另一个列表 Y 中，我们考虑下如何实现这个处理（图 10.1）。假设我们希望把整个处理作为一个完整不可分（原子性地）的过程执行。也就是说，在将从 X 删除的值到写入 Y 中的中间状态时，其他处理无法访问列表 X 和 Y。这种情况下该上什么样的锁呢？

图 10.1 完整不可分的处理



在线程安全的程序库中，程序员无需担心锁的控制方式，内部机制

可以保证使用锁后删除操作或写入操作不会被中间介入[①]。但这个锁无法保证将从 X 删除值往 Y 中写入时不会被中间介入。要防止中间介入，程序员必须用新的锁将这两个处理步骤包括起来，用 synchronized lock 把所有这些与 X 和 Y 读写相关的代码包括起来。这样就没能达到让程序员无需担心锁的控制方式的目的。那没有其他好方法了吗？

借助事务内存来解决

有一种叫做事务内存的方法可以解决这一问题[②]。这种方法把数据库中事务的理念运用到内存上，做法是先试着执行，如果失败则回退到最初状态重新执行，如果成功则共享这一变更（图 10.2）。它不是直接修改 X 或 Y，而是临时性地创建了一个版本对其进行修改，将一个完整不可分的过程执行完毕后才反映出最终的成果。

如果这个也拿试衣间来类比，情况就变得很奇怪了[③]。我们还是借之前的例子来进一步说明。在这两个处理的中间状态，如果有其他线程的读取要中间介入进来将会怎样呢？答案是即使有别的处理要介入进来，也只是创建了另一个临时的版本，对其作的修改不会反映到原来的数据上，在其他的线程看来数据的状态还是和执行 X 删除操作之前一样。这样就不存在任何问题。

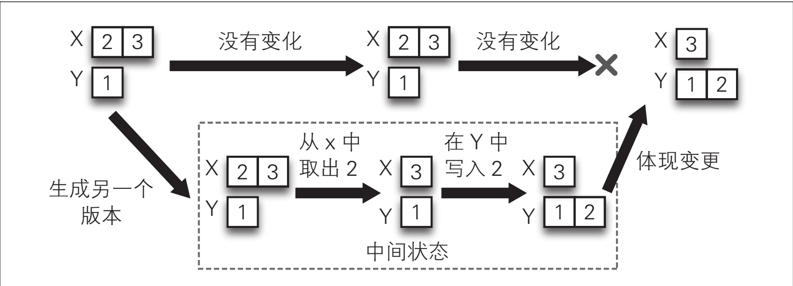
---

① 也有不使用锁且线程安全的程序库，要展开来讲话题就变复杂，这里进行简单化处理。

② Tim Harris, et al. “Composable Memory Transactions”, 2006. http://research.microsoft.com/pubs/67418/2005-ppopp-composable.pdf.

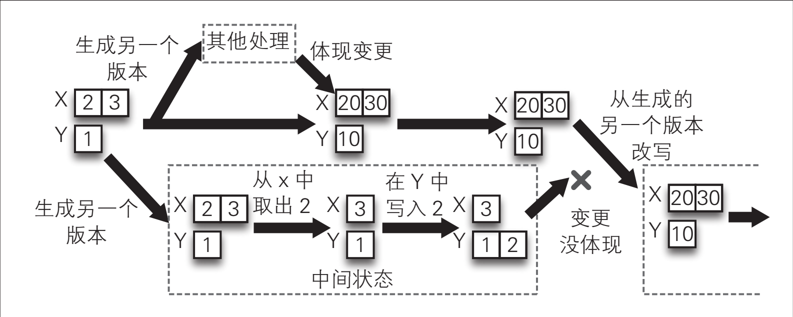
③ 如果硬要把它类比为试衣间那个例子，这个故事就变成：并行地存在进入试衣间的世界和没有进入试衣间的世界这样两个世界，当尝试进入试衣间有问题时，会折返到没有进入试衣间的世界中。这样的说法是完全没有现实意义的，当然这个可以在计算机世界中进行仿真。

图 10.2　创建另外的版本然后修改，处理结束后反映最终结果



假设有写入操作在中间介入进来（图 10.3），那么临时创建的版本就会被丢弃，重新回退到最初状态开始执行。这样一来，即使不上锁也可以顺利地进行并行处理。要注意的是，当写入的频率太高时，回退重新执行的操作就会多次执行到，这样会导致性能下降。

图 10.3　如果遇到别的写入操作介入则重新执行



事务内存的历史

硬件事务内存

1986 年，一家名为 Symbolics 的公司提出基于硬件实现的事务[①]。这家公司在 1981 年开始提供在硬件中安装了 LISP 语言的商用 LISP 机器——LM-2.

1986 年也是名为 MIPS R2000 的 CPU 诞生的一年。MISP 是基于

① Tom Knight, “An Architecture for Mostly Functional Language”, 1986.

RISC（Reduced Instruction Set Computer）这种旨在减少命令个数简化线路的设计方针制作出来的CPU。它和在硬件中实现LISP语言或事务这一方针相悖而行。个中原因非常复杂，最终，Symbolics在商业上也并未取得成功。

软件事务内存

在10后的1995年，一篇关于如何在软件中实现事务内存的论文发表了[①]。

又一个10年之后，微软公司于2005年发表了一篇关于使用Concurrent Haskell在软件中实现事务内存的论文[②]。

在此前后的几年间，很多编程语言都实现了软件事务内存的功能。比如，2004年IBM公司开发的X10和2006年Sun Microsystems公司开发的Fortress中都实现了这个功能。2007年，基于Java VM的Clojure发布。现在已经有一些介绍它的图书出版。

事务内存成功吗

事务内存这一技术在将来值得期待吗？未来会怎样没人知道。微软公司于2010年中止了面向.NET Framework平台搭载软件事务内存的实验。这么做的理由众说纷纭，但是认为能用到软件事务内存的杀手级应用缺失这样的悲观意见有很多[③]。

另外，据说后续的Intel处理器将搭载事物内存的部分功能。如若实现，届时对于硬件事务内存就可以轻松一试了。至于今后将如何进一步发展，我们只能拭目以待。

① Nir Shavit, Dan Touitou, "Software transactional memory", 1995.

② Tim Harris, et al, “Composable Memory Transactions”, 2006. http://research.microsoft.com/pubs/67418/2005-ppopp-composable.pdf.

③ “A (brief) retrospect on transactional memory”, http://www.bluebytesoftware.com/blog/2010/01/03/ABriefRetrospectOnTransactionalMemory.aspx.

10.6 小结

本章我们学习了并行处理这个重要概念。并行处理是长久以来人们一直为之烦恼的问题，至今也有很多人为之困扰。回顾历史，我们会发现，人们在共享→非共享→共享、协调→非协调→协调、硬件→软件→硬件这样两种对立观念中左右摇摆。不单看片面而是兼顾两面，在平衡中灵活运用或许才是最为重要的。

第 11 章

对象与类

本章我们来学习面向对象。

面向对象所指的内容因语言而异。

首先我们将在 SmallTalk 和 C++ 语言的比较中分析这种差异所在。

接着我们追溯历史来了解面向对象是如何诞生的。

最后，我们会逐渐深入挖掘面向对象的机制，来学习面向对象的具体内容。

11.1 什么是面向对象

内涵因语言而异的面向对象

语言中的用语并不是共通的，在不同语言中，同一个用语的含义可能会有很大差别。本书已经多次强调这一点。笔者认为，其中最为甚者就是面向对象这个概念了，至少有两位面向对象语言的设计者把面向对象一词用来表示两种完全不同的意义。尤其是关系到类型和继承时，两者的含义是完全相反的。

C++ 语言的设计者本贾尼·斯特劳斯特卢普在其著作 *The Design and Evolution of C++*[①] 中说道："class 是一种创建用户自定义类型的功能。"另外，他还在论文 [②] 中指出："Simula 的继承机制是解决问题的关键""面向对象程序设计是使用了用户定义类型和继承的程序设计"，从而对类和继承给予了正面肯定。

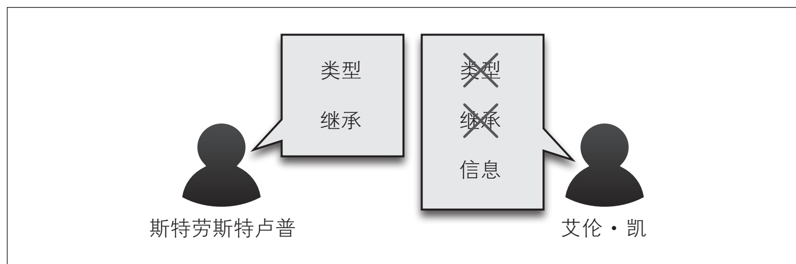
---

① 中文版《C++ 语言的设计与演化》由科学出版社于 2012 年出版，裘宗燕译。

② 1991 年举办的 1st European Software Festival（第一届欧洲软件节）上公开的论文"What is Object-Oriented Programming"（什么是面向对象程序设计）。在其网站（http://www2.research.att.com/~bs/homepage.html）的 research and popular papers 页面能查阅到。

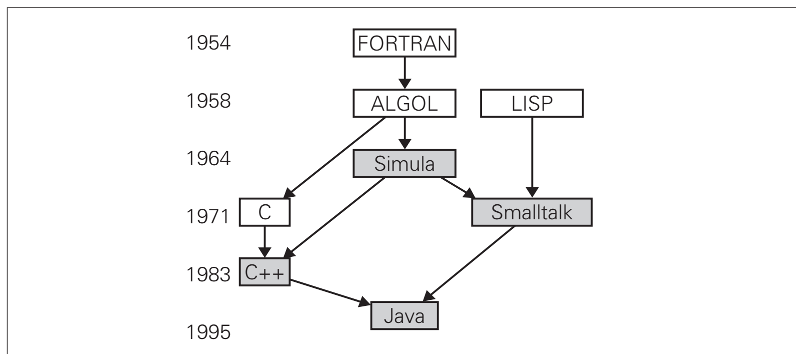
然而，“面向对象”这个词的发明者艾伦·凯（Alan kay，他同时也是 Smalltalk 语言的设计者）却持有不同的意见。当问到面向对象一词的含义时，他解释说：“我并不是反对类，但是我还没有见过不让我感到痛苦的包含类的系统”“我并不喜欢 Simula 里继承的做法”“通过不同状态的对象互相传送消息来通信的程序设计就是面向对象”①。由此可以看出，他对类和继承持否定立场（图 11.1）。

图 11.1 关于面向对象的相左意见



本书不打算纠结“到底何为面向对象”这一定义。我们将探讨面向对象发明的缘由以及动机。图 11.2 展示了本章提及的各编程语言大致的问世时间，供读者参考。

图 11.2 面向对象语言的大致问世时间及演化关系



※ 灰色的为面向对象语言

① Dr. Alan Kay on the Meaning of “Object-Oriented Programming”, http://www.purl.org/stefan_ram/pub/doc_kay_oop_en.

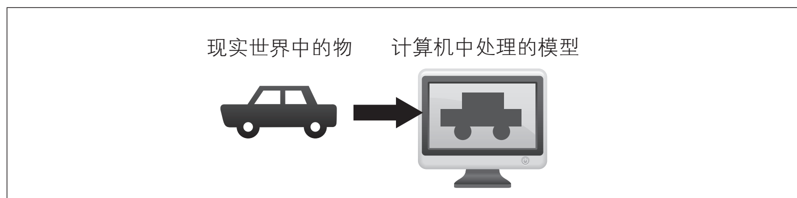
对象是现实世界的模型

我们是怎样理解世界的呢？我们将生活中遇见的事物总结为特定的“物”的概念，它们就是诸如桌子、椅子、银行贷款、公式、人、多项式、三角形、晶体管之类的东西。我们的思考、语言以及行动就是建立在指示、说明和操作这些所谓的“物”的基础之上。我们在用计算机解决问题的时候，有必要将现实世界中的“物”的模型在计算机中建立起来。

以上就是程序设计语言 ALGOL60 的设计者霍尔在 1966 年演讲的内容。[①]“物”的原文是 object（对象），“模型”的原文是 model（模型）。[②]现实世界中物的模型在计算机中应该怎样建立？怎么才能轻松地建立这些模型？经过多人研究探讨，面向对象这个概念隆重登场了（图 11.3）[③]。

这正是面向对象语言的设计者希望达到的目的！

图 11.3　现实世界中的物和计算机处理模型



① C.A.R. hoare, “RECOAD HANDLING” 1966, 1.Basic Concepts http://archive.computerhistory.org/resources/text/Knuth_Don_X4100/PDF_index/k-9-pdf/k-9-u2293-Record-Handling-Hoare.pdf “our thought, language, and actions are based on the designation, description, and manipulation of these objects, either individually or in relationship with other objects.”

② “RECOAD HANDLING” 1.Basic Concepts “we often need to construct within the computer a model of that aspect of the real or conceptual world...”

③ 尽管 ALGOL60 本身并不是面向对象语言，但通过阅读其扩展方案，我们可以了解在面向对象诞生以前人们在做怎样的思考。

什么是类

准备学习面向对象的朋友可能最初碰到的概念就是类吧。那么类到底是什么呢？这个问题，倘若一般地去回答就要陷入泥沼，也是一个危险的问题。因为在不同的编程语言中它有不一样的含义。至少在 C++ 语言里，类被定义为是“用户可自定义的类型”。但是正如我们在第 8 章学习到的，C++ 语言是静态类型语言，而 Ruby 语言和 Python 语言是动态类型语言。也就是说，在 Ruby 语言和 Python 语言中，类型一词指的内容和在 C++ 语言中是不一样的。在本章的剩余部分中，我们将会学习动态类型语言 Perl 和 JavaScript 中类的构造。

类，真有必要吗？抱有这样的疑问的人似乎挺多的。那么，类是有必要的吗？大部分语言的程序设计中，类并不是不可或缺的，但 Java 语言是例外。Java 语言“把类定义为部件，将其组装起来即是程序设计”。因此，在用 Java 语言编写程序时类是必要的。

其他诸如 C++、Python、Ruby 这样的语言，在编写程序时既可以使用类也可以不使用类。那是使用类好呢，还是不使用也可以呢？

这取决于要编写的程序。如果仅是小规模的程序，没必要使用类的情况居多。也有人认为，在多人分工协作编写的大型程序中，使用类来划分责任范围比较好。图形用户界面的编写中面向对象的特性似乎非常管用。比如设计一个按钮，需要有放置按钮的座标和按钮的宽、高等值，也需要有表达按钮按下时的动作的函数。将实现按钮所必需的这些要素统一到类中，编写程序就会变得简单起来。

11.2 归集变量与函数建立模型的方法

程序设计人员要归纳并建立模型。但归纳的方法各式各样，语言不同，选择也不尽相同。作为在 C++ 语言和 Java 语言中选取的方法，类非常有名。这里我们先来介绍几种类以外的方法。

第一个就是模块（module）。模块原本是一种将相关联的函数集中到一起的功能。在 Perl 语言中类似的功能被称为包（package）。Perl 语言在引入面向对象时，采用了把用来归集函数的包和用来归集变量的散列（hash）绑定在一起的方法。

第二个方法是把函数和变量放入散列中。这是 JavaScript 等语言采用的方法。

第三个是闭包（closure）。我们会讲解这种使用函数执行时的命名空间来归集变量的方法。这种方法主要在函数式语言中使用。

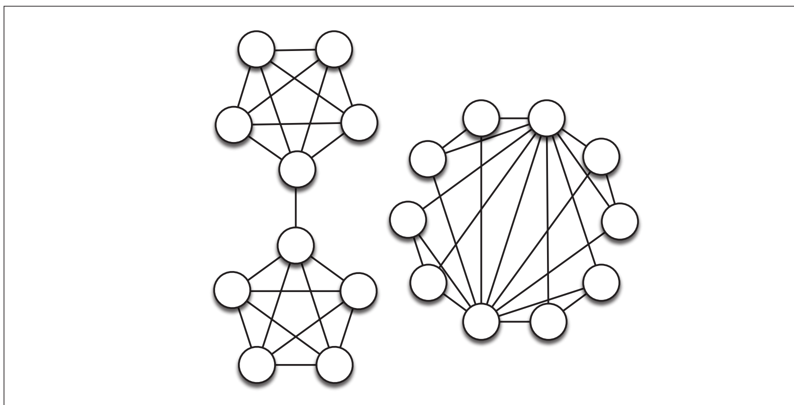
之后，我们就类展开探讨。

11.3 方法 1：模块、包

什么是模块、包

一个程序中有多个构成要素，它们之间存在相互作用。如图 11.4 所示，每个图形都由 10 个要素和 21 个相互作用关系构成。那么其中哪个更便于理解呢？

图 11.4　哪个更容易理解



以前的程序设计中，函数和变量是处于同等地位并且散布在程序中的。不管哪个函数、哪个变量，在程序的任何位置都能访问。但是，为了设计出更容易理解的程序，于是出现了互相有紧密联系的组。与使所有的元素都和其他元素保持同等的作用相比，把关联性强的几个元素归集在一起的方式更有助于理解。

1978 年左右开发出来的 Modula-2[①] 导入了模块的概念，显示地表达了这种关联性强的几个函数和变量的组合。至今很多编程语言都延续了这个机制。Python 语言和 Ruby 语言继续把它称为模块，而 Java 语言和 Perl 语言则把它称为包。

模块是一种归集的方法。既然如此，那是不是可以借助它来归集变量和函数，从而设计现实世界中的物的模型呢？

用 Perl 语言的包设计对象

Perl 语言中的包是一种能将函数和变量打包并命名的一种功能。接下来，我们使用这种包来设计现实世界的物的模型。

这里我们要设计的是交通流量调查使用的和日本野鸟协会用来统计鸟类数量的计数器，想必大家对这个都耳熟能详了。这种计数器实现了按下按钮显示数字增加，按下复位按钮显示数字归零的功能。其中有“按下按钮”和“按下复位按钮”两种操作，以及“显示的数字”这一个值。

用Perl语言的包设计计数器

```
{
    package Counter;
    my $count = 0;
    my $name = "麻雀";

    sub push{
        $count++;
        print "$name: $count只\n";
```

① Modula-2 语言的设计者尼古拉斯·沃斯（Niklaus Wirth）曾经说过，Modula-2 的祖先是 Pascal 语言和 Modula 语言。Modula-2 从后者继承了名字、模块这一重要的概念以及系统和现代的语法结构，其余的特点则几乎都是从 Pascal 语言继承而来的。请参考 Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, 1989, p.143。

```
    }
    sub reset{
        $count = 0;
        print "$name: 重置\n";
    }
}

Counter::push;                #-> 麻雀: 1只
Counter::push;                #-> 麻雀: 2只
Counter::push;                #-> 麻雀: 3只
Counter::reset;               #-> 麻雀: 重置
Counter::push;                #-> 麻雀: 1只
```

如此便大功告成了！这里实现了一个在计算机上称为 Counter 的模型，执行一次 Counter : : push 数值加 1，执行一次 Counter : : reset 数值归零。这和在野鸟数量统计时按下按钮数字加 1，按下复位按钮数字归零是一样的动作。这样就在计算机中实现了现实世界中物的模型。

光有模块不够用

然而，光有模块是不够的。我们回到日本野鸟协会的计数器的例子。假设这一次要统计麻雀和乌鸦分别的数量，并且买了两个相同的这种计数器，分别称为 A 和 B。计数器 A 和计数器 B 有相同的型号、相同的功能，因此属于同一种类型。但这不是同一个东西。按下计数器 A 的按钮，表示的数字会加 1，而计数器 B 的数字却不会改变。

函数和模块，每定义一次就对应计算机上的一个新的实体。而现实世界中常常有相似的事物同时存在多个的情况。怎样才能将这种现实世界的构造在计算机上用模型表现出来呢？把 Counter 包选定、复制，分别创建新的包 Counter A 和 Counter B，是不是就可以呢？当然，这样做是可以实现的。但有谁会愿意为表达 10 个相似的东西，对一段代码复制 10 次然后去维护 10 段完全相同代码呢？估计大家都不喜欢。我们需要一种更加便捷的方法[①]。

① 一言以蔽之，就是想要创建多个实例。

分开保存数据

我们来整理一下面临的问题。函数和模块的定义和实体是一对一的。按下按钮数字增加 1 的操作对于计数器 A 和计数器 B 都是相通的。从这个意义上来说，只需要一个计数器就足够了。

但是计数器的值对于计数器 A 和计数器 B 是不同的。计数器 A 的值变化了，也不能影响到计数器 B 的值。从这个意义上来说，又确实需要多个计数器。

也就是说，如果有方法能分开保存数据就可以了[①]。

向参数传递不同的散列

Perl 语言的语言处理器本身就带有我们在第 9 章学习的字典（名字与值的对照表）创建功能。Perl 语言中称之为散列。那么把散列用于存储数据会怎么样呢？关于函数的定义我们不做任何改变，继续使用包作归集。在函数调用时，把散列作为实参传递给被调用函数。

```
Perl
{
    package Counter;
    sub push{
        my $values = shift;     ❶
        $values->{count}++;     ❷
        print "$values->{count}只\n";   ❸
    }
}

{
    # ❹创建散列
    my $counter = {"value" => 0};
    my $c2 = {"value" => 0};

    # ❺将散列传递给参数
    Counter::push($counter);    #-> 1只
```

① C++ 语言中可以实现一个用户定义类型对应多个变量的形式，这一点将在后面介绍。

```
    Counter::push($counter);     #-> 2只
    Counter::push($c2);          #-> 1只
    Counter::push($counter);     #-> 3只
    Counter::push($c2);          #-> 2只
}
```

❹句中的 { "count" => 0} 部分定义了一个键名为 "count" 对应的值为 0 的散列。这里定义了两个不同的散列 $counter 和 $c2 作为数据的保管场所，然后将它们作为参数传递给 Counter 包中的 push 函数（❺）。push 函数中的❶句的意思是取出实参中的一个数值，代入 $values 中。随后，计数器加 1（❷），并打印输出（❸）。从这里我们可以看出，两个计数器是分别独自增长的，这便和现实世界中的计数器一样了。$counter 的值不会干涉到 $c2 的值，这是因为两者是不同的对象，各自维持自身的状态。

把初始化处理也放入包中

然而，在这一实现方式下每创建一个新的计数器时，程序员都必需编写 { "count" => 0}。也就是说，程序员必须记住如何初始化这些值。这不是一种好的设计方式。对于这种定型的操作，相比人为地记住并加以注释说明，用代码去表现这种操作的方式显然要更好。这样就促成了初始化操作的函数化，并把它放入包中。

我们马上来看一看这是如何做到的。下面的代码中增加了一个名为 new 的函数。

```
Perl
{
    package Counter;
    sub new{
        return {"value" => 0};
    }
    sub push{
        my $values = shift;
        $values->{count}++;
        print "$values->{count}只\n";
```

```
    }
}

{
    # 把初始化处理放入包中
    my $counter = Counter::new;
    my $c2 = Counter::new;

    # 把散列传递给参数
    Counter::push($counter);  #-> 1只
    Counter::push($counter);  #-> 2只
    Counter::push($c2);       #-> 1只
    Counter::push($counter);  #-> 3只
    Counter::push($c2);       #-> 2只
}
```

把初始化操作定义为名为 new 的函数，这样程序员在每次创建新的计数器时，只要编写代码 Counter : : new 就可以实现了。在语言功能上并不是强制要求这样来写，但这样写的好处是使得程序变得更加简明清晰。这也可以说成是一种设计模式。像 new 这样创建新的对象的函数被称为构造函数（constructor）[①]。

把散列和包绑定在一起

到目前为止，我们要实现的目的已经能达到了，但 Counter : : push($counter) 看起来总觉得过于冗长。$counter 本身就是为了与 Counter 包配套使用而创建出来的，每次在使用包时必须一一指定 Counter : :，这是件很麻烦的事情。应该可以有更加轻松的实现方式。

我们考虑是否可以让语言处理器记住这个散列是和 Counter 包配套使用而创建出来的这一信息。为达成此目的，Perl 语言引入了 bless（祝福）这一概念，并提供了 bless 函数，它可以把散列和包两者绑定在一

① 我们已经多次强调，语言不同术语的含义会有差异。比如在 Java 语言中构造函数（constructor）的含义更为有限。这个例子在 Java 语言中的表达是 Counter.newInstance( ) 这一方法，但它不叫做构造函数而是叫做工厂方法（factory method）。

起，创建一个 blessed hash。

在前面的程序末尾补充以下几行代码。

```perl
{
    my $counter = {"value" => 0};
    print "$counter\n";
    #-> 输出HASH(0x1008001f0)❶
    # 这是没有被bless的散列

    # 把散列和包绑定一起
    bless $counter, "Counter";
    print "$counter\n";
    #-> 输出Counter=HASH(0x1008001f0)❷
    # 这是被bless 的散列

    $counter->push;  #-> 1只 # 轻松地使用箭头运算符!
    $counter->push;  #-> 2只
}
```

在被 bless 之前创建的 $counter 显示出来是❶的样子，而使用了 bless 函数之后创建的 $counter 显示出来是❷的样子。散列和 Counter 包已经绑定配套，变成了 blessed hash。

并且，在被 bless 的散列使用箭头运算符 -> 后，程序会到与之绑定的名字的包中寻找相应的函数，把该散列传递给该函数并调用[①]。在这个例子中，$counter->push 一句执行时，会到与 $counter 绑定在一起的名为 Counter 的包中寻找名为 push 的函数，把 $counter 作为参数传递给这个函数并调用它。

通过把数据的保存场所和数据操作的集合（模块）绑定在一起，看上去就变成一个整体，非常清晰。而绑定操作本身也是一个定型的操作，故把它一起放进 new 函数中。

```perl
sub new{
    my $class = shift;
    my $values = {count => 0};
```

① 严格来讲，bless 的不是散列而是散列的引用。

```
    bless $values, $class;
}
```

这样一来，通过 $counter = Counter : : new 就创建了一个新的计数器，通过 $counter->push 就实现了按下这个计数器按钮的功能，这样看上去十分简洁清晰。

综上，通过使用一系列的方法，我们实现了野鸟计数器的整体建模，可以创建对象 (Counter : : new)，也可以为这个对象命名 (my $counter = ...)，还可以操作这个对象 ($counter->push)。于是，在计算机中创建现实世界中的物的模型这一目的就很好地达成了。

11.4 方法 2：把函数也放入散列中

first class

Perl 语言中使用包把多个函数归集在一起。接下来，我们要介绍的是 JavaScript 语言中使用的另一种归集方法。这种方法把函数也放入散列中。

大家使用的编程语言大多应该可以把字符串赋值给变量。也可以把它作为函数的参数传递或作为函数的返回值返回。或许你会觉得这是理所当然的事情。事实上，并非所有的语言都如此。比如，FORTRAN 66 中就不能把字符串赋值给变量。另外，C 语言中就不可以把数组作为参数来调用函数[①]。

像这种不受限制，可以赋值给变量，也可以作为函数的参数传递，又可以作为函数的返回值返回的值被称为 first class 的值。这好比不受任何歧视的一等公民（first-class citizen）。最新出现的一些编程语言中，如 Java 语言、Perl 语言和 Python 语言，字符串就是 first class 的值。

① 看起来像是把数组或者字符串传递给了函数，事实上只是指向数组开头位置的指针。

在 JavaScript 语言中，函数也是 first class 的值。函数可以被赋值给变量，可以作为函数的返回值返回。接下来，我们来考察利用这一特征可以实现哪些功能。

把函数放入散列中

JavaScript 中的散列是用如下语句定义的。

```
JavaScript
{count: 0, name: "麻雀"}
```

这里像 0 和 " 麻雀 " 这样表达值的部分，可以放入函数 function( ){ ... }(图 11.5)。

▌图 11.5　counter 是散列

counter	
count	0
name	麻雀
push	function(……)
reset	function(……)

接下来，我们来看这是如何实现的。下面的代码和 11.3.2 节中介绍的内容大体相同，实现了野鸟计数器的功能。

```
JavaScript
var counter = {
    count: 0,
    name: "麻雀",

    push: function(){
        this.count++;
        console.log(this.name + ": " +
                    this.count + "只");
    },
    reset: function(){
        this.count = 0;
        console.log(this.name + ": " +
                    "重置");
```

```
    }
}

counter.push(); //-> 麻雀: 1只
counter.push(); //-> 麻雀: 2只
counter.push(); //-> 麻雀: 3只
counter.reset();//-> 麻雀: 重置
counter.push(); //-> 麻雀: 1只
```

在 Perl 语言中是通过包来实现的，而在 JavaScript 语言中是通过散列实现的。此外有没有别的不同之处呢？乍一看，还有关键字 this 的使用这一区别。this 是一个限定词，在函数 my_method( ) 与对象 obj 绑定之后通过 obj.my_method( ) 的形式调用此函数时，this 用于在 my_method( ) 函数中引用 obj 对象本身。前面的 Perl 语言的例子中，我们用 my $value = shift 语句来显式地获取参数，而 JavaScript 语言中却是隐式地借助了 this 这一变量来表示。上述的例子中通过 counter.push( ) 语句来调用函数，因此 push 函数中使用的 this 变量指的就是 counter。所以这个例子中把 this 替换成 counter 程序也是可以正常执行的。

这个例子和使用包的例子都有一个共通的问题，就是只能创建一个计数器。但是这里要创建多个计数器也不是困难的事情。下面我们就来展示这是如何做到的。

创建多个计数器

为了创建多个计数器，我们需要定义一个散列初始化的函数。编写这样一个函数是出乎意料的简单，只需要把创建散列的代码挪到 makeCounter 函数中就可以。这样就达到了和 Perl 语言中使用包一样的效果[①]，即：

- 可以创建多个对象
- 外观上是一个完整不可分的整体
- 无需人为记住初始化方法

① 因篇幅所限，这里省略了 reset 方法。

```JavaScript
function makeCounter(){
    return {
        count: 0,
        push: function(){
            this.count++;
            console.log(this.count + "只");
        }
    }
}

var c1 = makeCounter();
var c2 = makeCounter();
c1.push(); //-> 1只
c2.push(); //-> 1只
c1.push(); //-> 2只
```

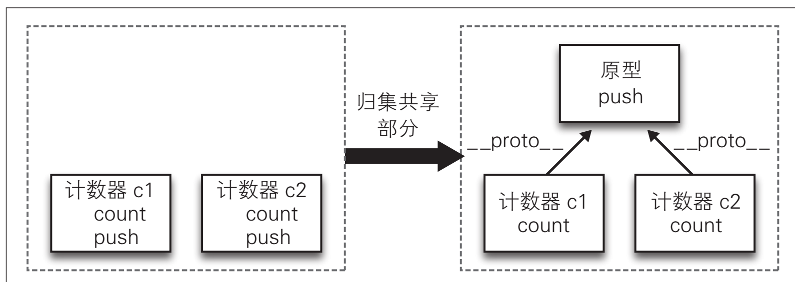
把共享的属性放入原型中

刚才的代码中，每次创建计数器时都会重新定义一个 push 函数。用以下语句确认 c1.push 和 c2.push 是否相同时，返回值是 false。这意味着两者是不同的。

```JavaScript
console.log(c1.push === c2.push); //-> false
```

这是什么原因呢？在每次调用 makeCounter 时 push: funtion( ){...} 会被执行，定义一个新的函数。也就是说，创建 100 个计数器，就会有 100 个内容相同的 push 函数被定义。如果内存和 CPU 可以无限供应时，这不是什么问题。但现实没有这么美好。如果能把 push 函数这样所有计数器都共享的属性归集起来，个别计数器只是对其做引用，这样应该能节省内存和时间（图 11.6）。

图 11.6　把共享的属性归集起来



然而，归集起来后就要记得放入了哪里，使用的时候还要显式地指示出来，这样很麻烦。把共享的内容放在别的对象中，那就必须记住这一内容放在哪个对象中，人工来记忆这些是令人不快的。如果语言处理器能代劳并做出正确的判断就好了。

原型的操作

为解决这一问题，JavaScript 语言引入了原型的概念[①]。当向一个对象查询 x 的值时，如果这个对象自己知道就自己给出答案。如果不知道，它会去查询它的原型再给出答案。下面的代码中，obj 对象就不知道 x 的值是多少。

```
JavaScript
obj = {}
obj.__proto__ = {x: 1}

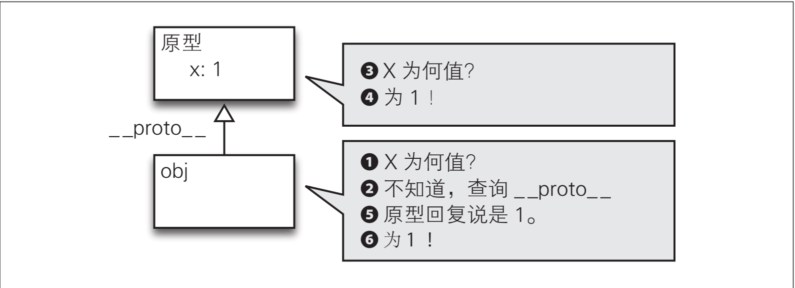
console.log(obj);             // -> {}
console.log(obj.__proto__); // -> { x: 1 }
console.log(obj.x);           // -> 1
```

在查询 x 的值时 (obj.x)，obj 转向它的原型 (obj.__proto__) 才给出答案（图 11.7）[②]。

① 这里介绍的是用委托的方式实现原型这一概念的情景，不同语言中也可以在实例化时通过负责实现。至于这种方法中当实例化后原型发生了变更会怎样，在不同语言中是存在差异的。

② 通过 __proto__ 这个名字访问原型并不是标准功能，根据处理器的不同会有不同的可能性。JavaScript 1.8.1 开始引入了 Object.getPrototypeOf (object) 的表达方式。

图 11.7 查询 obj.x 时发生的事情



用 new 运算符实现高效表达

除此之外，JavaScript 语言还引入了一种使得原型处理的表述更加方便的运算符。在函数 f 前使用 new 运算符后会执行以下四个操作：

- 创建新的对象 x
- 新创建的对象 x 的原型变为函数 f 的原型
- 把新创建的对象 x 传给 this，执行函数 f 的内容
- 返回对象 x

```
JavaScript
var Counter = function() {
    this.count = 0; //                          ❶
}

Counter.prototype.push = function(){ //         ❷
    this.count++;
    console.log(this.count + "只");
}

var c1 = new Counter(); //                      ❸
c1.push(); //-> 1只
c1.push(); //-> 2只
var c2 = new Counter();
console.log(c1.push === c2.push) //-> true //   ❹ 相同
```

这段代码中，首先通过❷句在 Counter 的原型中追加了一个名为 push 的新的函数。在随后的❸句执行了上述四个操作。首先创建了一个

空的对象，然后将这个对象的原型设为 Counter 的原型。这个原型中定义了 push 函数。接下来，把这个对象传给 this，再调用 Counter。Counter 中包含❶句的内容。执行这些内容会往这个对象中追加一个名字为 count 值为 0 的属性。最后，返回一个原型中包含 push 函数的、带有 count 属性的对象。不经意间我们就实现了图 11.6 右边部分的结构。❹句返回值是 true，由此可知 push 函数已经是共享的了。

这就是面向对象吗

在 Java 语言中学过面向对象的读者可能会心里犯嘀咕，觉得到现在为止讲的内容远远不够。有些人甚至可能要发怒，质问为什么关于类的话题还没有任何涉及。

笔者认为，不知道面向对象的读者当中很多读者也不知道何为类。只是大家看的此类书籍都是从类开始讲面向对象的。

的确，Java 语言是一门编程必从类开始的语言。笔者也深知，在教授 Java 语言程序设计时，不先详细讲解并使之学会使用类是不行的。然而，类并不是从程序设计语言诞生之日起就存在的。它充其量只不过是几十年前某个人出于提高便利性的需要，试着创作出来的东西而已。尽管如此，我们总是被提醒类的存在并被鼓励去使用它，但又不太明白类为什么是必要的。

类的存在只不过是因为人们觉得有了它编写程序会更方便些，而约定的一种事项。它并不是什么物理法则或宇宙真理，仅仅是人们的一种约定而已。所以，为了理解为什么会有这样一种约定，我们需要考虑语言设计者的意图。

11.5 方法 3：闭包

什么是闭包

说到闭包（closure）这个概念，想必很多人一时也说不出究竟何为闭包。它是创建具有对象性质的事物的一种技术。

很多语言都支持定义带有某种状态的函数。比如，可以定义像计数器一样每调用一次显示的数字加 1 的函数。我们用 JavaScript 语言来实现一下。

```
JavaScript
function makeCounter(){
  var count = 0;
  function push(){
    count++;
    console.log(count);
  }
  return push;
}

c = makeCounter();
c();                    //-> 1
c();                    //-> 2
c();                    //-> 3
```

这段代码中，函数 makeCounter 中定义了变量 count 和函数 push，并返回函数 push。然后，通过调用函数 makeCounter 将返回值赋给变量 c，然后调用它三次。每调用一次，显示的值就加 1。这是怎么做到的呢？函数 makeCounter 首先创建了一张名字和值的对照表，把变量 count 的值设为 0。然后定义了函数 push，并将其返回。函数 push 将其被定义时的对照表一同带出 makeCounter 函数。随后，每当被调用时，函数 push 在被定义时的对照表的值就加 1。

事实上，并没有所谓闭包的特殊的语法结构。如果有一种语言[①]，它可以在函数中定义函数，有允许嵌套的静态作用域，并且可以把函数作为返回值传递给变量，那么它只要通过函数的嵌套就可以实现带有某种状态的函数。

为什么叫做闭包

看到闭包这个名字，总有一种什么东西被严实地包裹起来了的感觉。为什么会叫做闭包呢？某 Standard ML 的教材上做了如下解释。

> 为什么把这称为闭包？一个包含了自由变量的开放表达式，它和该自由变量的约束环境组合在一起后，实现了一种封闭的状态。
>
> —— Åke Wikström, *Functional programming using standard ML, Prentice-Hall*, 1987.[②]

拿上面一段代码来讲，函数 push 使用了变量 count，然而该变量并不是在函数 push 中定义的。这种变量被称为自由变量。函数 push 就是一个包含了自由变量的开放函数。而函数 makeCounter 的对照表中为 0 的值和为 count 的名字结合在了一起。这种给值绑定一个名字的操作叫做（名字）约束。这样开放函数 push 和 makeCounter 的对照表组合配套之后，无需在这以外的作用域中寻找变量的定义，从而达到了某种完备的状态。通过这样表现出一种封闭的属性。

11.6 方法 4：类

那么，何为类？它是语言设计者在特定语言中规定的一些称呼，具有多种定义。

① 大部分的函数式语言和 JavaScript 语言以及 Python 语言都满足这一条件。Perl 语言和 Ruby 语言经过一些处理也可以符合这一要求。

② 原文是“The reason it is called a ‘closure’ is that an expression containing free variables is called an ‘open’ expression, and by associating to it the bindings of its free variables, you close it”。

霍尔设想的类

首先来看原始形态的类是怎么定义的。1965 年霍尔在 ALGOL 的扩展方案中写道："基于便利性的考虑，现在世界中的物（objects）通常被分为几种独立的分类，而某种分类的物如能进一步分为更细的类别（subclasses）就更方便了[①]。"

从这段话能看出来，他所谓的类就是分类。它和我们日常说的 economy class（经济舱）中的 class 以及分组对抗赛中的分组属于相同的用法。现在 C++ 语言和 Java 语言中使用的 class 一词被追加了很多意思变得复杂起来，但最初都是分类的意思。现在所说的类的继承以及类是像做鲷鱼烧一样创建实例的鲷鱼烧的模具，这些说法都是后来才出来的概念。

C++ 语言中的类

十年之后的 1979 年，本贾尼·斯特劳斯特卢普开始了 C with Classes 的开发。这也就是之后的 C++ 语言。C++ 语言中类的概念是以 Simula 这种仿真用语言[②]中的类为参考，方便用户定义类型（type）而设计出来的。斯特劳斯特卢普是这样解释的：

> 类是类型。这是 C++ 语言中极为重要的一种思想。既然 C++ 语言中类的意思就是用户定义的类型，那为什么不把它叫做 type 呢？我之所以选择使用 class，是因为实在不喜欢不断地创造新词，才选用 Simula 语言中 class 这个谁都不至于困惑的词。
>
> ——斯特劳斯特卢普《C++ 语言的设计与演化》

能让用户定义新的、操作起来如同 int 和 float 这样内置型的类型，这是 C++ 语言中引入类的目的。C 语言中也有能将各种变量归集到一起定义新的结构体的功能。C with Classes 中的类最初就是 C 语言中的结构体。之后，斯特劳斯特卢普把他认为好用的功能陆续加入了进来。其

① 这是从"RECOAD HANDLING"第 3 页和第 15 页翻译而来的。

② Simula 是一种专注于仿真的程序设计语言。在这种语言中，由类创建的对象会在协调的多线程模式下，像 Erlang 语言的进程一样执行并行处理。

中之一就是功能说明的作用。

功能说明的作用

对于 C++ 语言来讲，类 (= 类型) 也是功能说明的一种表现形式。也就是说，类起着一种作用，它可以声明对象具有哪些方法和不具有哪些方法。如果程序调用了不存在的方法会导致编译错误。这也是 Smalltalk 语言和 C++ 语言一个很大的区别[①]。

在 Smalltalk 语言中，方法的调用就是向对象传送一个消息，告诉它去执行某某名字的方法。至于对象接收到这个消息后如何响应（执行什么操作还是导致错误还是不作任何处理），这可以由接收方的对象自由决定。Smalltalk 语言的设计者同时也是面向对象一词的发明者的艾伦・凯认为，这样的自由度是面向对象的重要元素之一。

类的三大作用

大家是不是觉得类这个概念很复杂？的确如此。这是因为 C++ 语言和 Java 语言的类具有以下几个作用：

❶ 整合体的生成器
❷ 可行操作的功能说明
❸ 代码再利用的单位

Perl 语言和 JavaScript 语言的说明主要集中在生成器的功能方面。类是制作鲷鱼烧用的模具这一说法讲的就是这个作用。

11.6.2 节介绍的是类的功能说明的作用。Java 语言中可以定义专门实现这一功能的接口。而动态类型语言不太重视这种作用。

第 12 章我们将探讨作用代码再利用的单位的类的作用，即继承[②]。

---

① 请参考：斯特劳斯特卢普 , “What is Object-Oriented Programming? (1991 revised version)” , 1991, p.15. http://www.stroustrup.com/whatis.pdf

② 当然，并不是说现在的 Perl 语言和 C++ 语言中类作为代码再利用的单元的作用消失了，只是类在不同语言中发挥的这种作用有强弱之别。另外，至少在早期 Perl 语言的包中并不具有继承功能。

11.7
小结

本章我们学习了面向对象产生的原因，它是为了创建现实世界中物的模型而产生的。同时，我们了解到，不同语言中类的实现方式以及面向对象这一术语本身的意义也是不同的。

第12章

继承与代码再利用

本章我们来学习继承。

继承这一机制也是因语言而异。

我们将在不同语言的继承机制的比较中来认识它们各自的长处与短处。

12.1 什么是继承

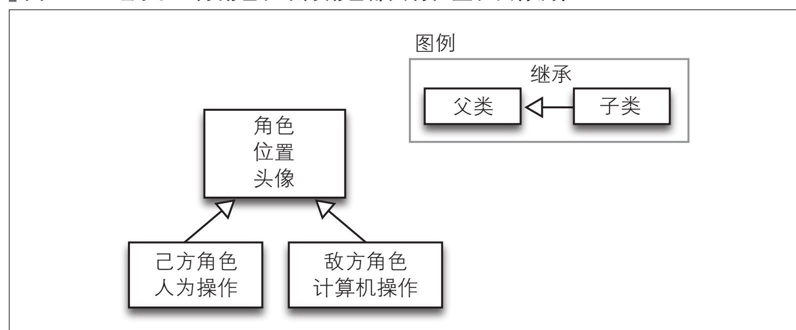
如第 11 章所述，类的最基本作用是分类。同一类别的事物具有共同的属性。即使将分类进一步细化，这些属性还是会被继承下去。

假设你要设计一个射击类的游戏。游戏中出场的角色为具有位置和头像的模型，并且进一步可以分类为由人操纵的己方角色和由计算机操纵的敌方角色。当然，己方角色和敌方角色都一样，具有位置和头像的属性。可见分类进一步细化后属性得到了继承。

己方角色和敌方角色在实际实现中，分别声明各自的位置和头像属性将是一个重复工作。如果类中已经声明的属性能在对其进一步细分的子类[①]中自动传承，这种语言的功能是非常好的。因此，继承产生了（图 12.1）。

① 子类（subclass）是从其他类中继承而创建的类。类 Y 从类 X 继承得到，就可以说类 Y 是类 X 的子类。子类的反义词是父类（superclass）。Y 是 X 的子类同时也意味着 X 是 Y 的父类。

图 12.1　继承：己方角色和敌方角色都具有位置和头像属性



如能使用继承实现方式的功能，那编程实现不就变得轻松很多吗？出于这种考虑，继承的使用变得广泛起来。同样是继承，考虑问题的方法和使用方式不尽相同。

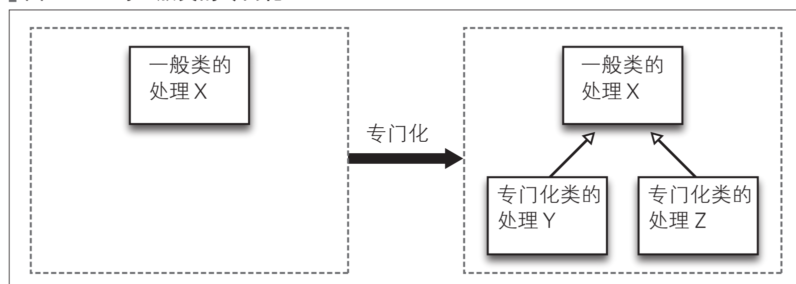
继承的不同实现策略

继承的实现策略大体可以分为三种。

一般化与专门化

第一种策略是在父类中实现那些一般化的功能，在子类中实现那些专门的个性化的功能（图 12.2）。其设计方针就是子类是父类的专门化。在更细致的层面上，它和分类意义是相近的，这和 class＝分类的想法一致。它让人很自然地意识到子类就是父类的一种。

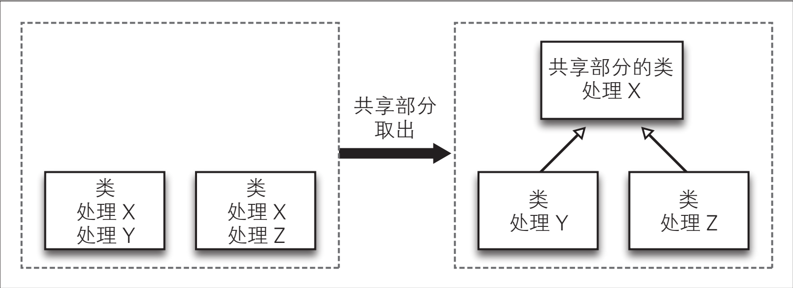
图 12.2　对一般类的专门化



共享部分的提取

第二种策略是从多个类中提取出共享部分作为父类（图 12.3）。它和一般化与专门化的考虑很不一样。对于子类是否为父类的一种，它的答案是否定的。这种提取出共享部分的设计方针是习惯了函数的一种考虑问题的方法。

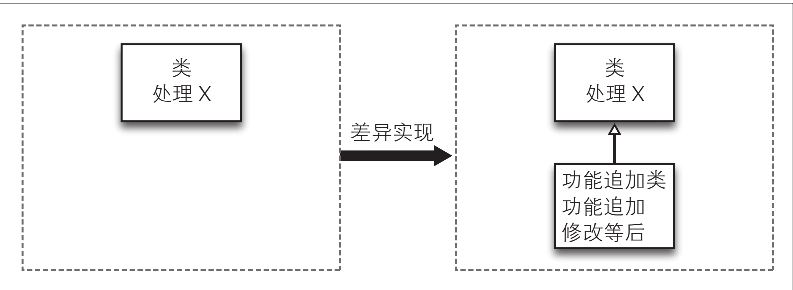
图 12.3　从多个类中提取共享部分



差异实现

第三种策略认为继承之后仅实现有变更的那些属性会带来效率的提高（图 12.4）。它把继承作为实现方式再利用的途径，旨在使编程实现更加轻松。的确有很多这样的情况。但这些情况下通常子类都不是父类的一种。

图 12.4　继承已有的类并实现差异部分



继承是把双刃剑

使用方法多意味着继承这种机制有很高的自由度。这和我们在第 4 章中接触到的 goto 语句有点类似。滥用 goto 语句可能造成代码理解困难，因此对其使用应加以限制。与此类似，对继承的使用也有相同的考虑，应该对其加以限制。尤其是第三种使用方法——继承已有的类并实现差异部分，这种编程风格会造成多层级的继承树，很容易导致代码理解困难。

多层级的继承树是如何导致代码理解困难的呢？假设某个对象具有方法 X，这一方法的定义在哪里呢？在这个类中，它的父类中，还是它的父类的父类中？要知道答案就需要追溯继承关系检查多个类[①]。此外，如果你要修改某个方法，这将影响到所有的子类，以及所有子类的子类。影响范围越广，就越难确定这种修改会不会带来什么问题[②]。

这个和我们在第七章中谈到的动态作用域的问题非常相似。人的理解能力是有限的，影响范围太大的话就理解不了了。影响范围小理解起来就轻松。通过使用继承实现代码的再利用，对于编写程序来说代码编写量减少了，工作变轻松了。但是反复使用继承后代码的影响范围变大了，理解起来也困难了。因此，为了保证理解的简易性，就要防止继承树的层级过多。

里氏置换原则

在第 6 章我们提到过 CLU 语言，它的设计者芭芭拉·利斯科夫（Barbara Liskov）等人在 1987 年提出一种原则——里氏置换原则，这一原则现在常常在创建子类时作为注意点被提及。

这个原则可以表述为：假设对于 T 类型的对象 x，属性 q(x) 恒为真。如果 S 为 T 的派生类，那么 S 类型的对象 y 的属性 q(y) 也必须恒

① 现在有很多集成开发环境都可以承担这种繁重的任务，这说明为使编程这项工作更加轻松，相关工具也得到了进化。

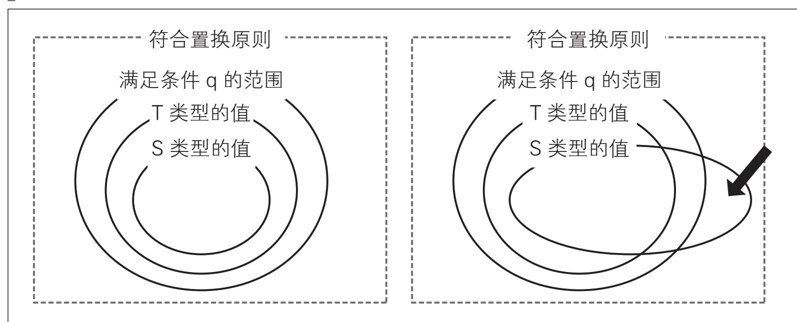
② 这个问题不是单靠在程序设计上下工夫就能解决的，同时也可以把大量的检查工作交给计算机去完成来帮助解决，这叫回归测试。解决问题的方法总是有很多种。

为真[①]。

这句话换种表达就是，对于类 T 的对象一定成立的条件，对于类 T 的子类 S 的对象也必须成立[②]。

语言表达可能比较难理解，我们用图来说明（图 12.5）。符合置换原则的情况如图左边所示，类 T 的所有对象都满足条件 q。并且类 T 的子类类 S 的所有对象都满足条件 q。从图中可以看出 S 是 T 的子集，这是很自然的。

图 12.5 里氏置换原则



打破置换原则的情况如图右边所示。箭头指向的部分有不满足条件 q 的 S 类型的值。这边 S 不是 T 的子集。比如在 Java 语言中，如果 S 是 T 的子类，那么可以将类 S 的对象传递给类 T 的类型的变量。即在类型系统中 S 是 T 的子集。然而在现实情况中，最初所有的 T 都满足条件 q，但 S 继承了 T 之后就出现了不满足条件 q 的类 T。因此，为了保证类的继承关系和类型的父子关系这两种关系之间的一致性，有必要遵守这一原则。

---

① 芭芭拉・利斯科夫，周以真，“A behavioral notion of subtyping”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol.16, Issue 6, ACM, 1994, pp.1811-1841. 在 1987 年的演说“Data abstraction and hierarchy”中，她们最早提出了这个思想。这里的解说针对的是 1994 年的确定性说法，其原文表述是：“Let ø(x) be a property provable about objects x of type T. Then ø(y) should be provable for objects y of type S where S is a subtype of T.”。

② 这里简单化处理把子类和 subtype 等同视之。至少在 C++ 语言和 Java 语言中这样理解应该没有问题。

这一原则也可以表达为继承必须是 is-a 关系。把子类 S 的所有对象都看作是父类 T 的对象而不会有任何问题，必须要做到这一点。

这一约束条件是非常严格的。当要继承某种类时，需要考虑该类是否可以被继承。假设继承的时候考虑的属性可以使里氏置换原则成立。但是在随后的程序编写过程中，需要的属性可能会越来越多。随着属性的增加，置换原则就有可能被打破。是在设计阶段就把所有属性列出来，只有当置换原则绝对不被打破时才去继承呢？还是在开发阶段如果发现新的属性就放弃类的继承呢？不管哪种方式都很费劲。

12.2 多重继承

我们了解了保证类的继承和类型的机制之间的一致性的难处。类型相当于我们在第 11 章中学习的类的三种作用之中的可行操作的功能说明。

另一方面，发挥类作为代码再利用单元的作用时，类型和类就是分类这种观点有时具有适得其反的效果。尤其对于动态类型语言这样不太重视类型的语言。

本节我们将深入探讨使用类来实现代码再利用的方法。

一种事物在多个分类中

把文件放进不同文件夹进行整理时，常常会有这个或者那个文件该放入哪个文件夹的烦恼。这是因为现实世界中的物并不仅仅属于某一种分类。

举个具体的例子，假设要把公司职员进行分类。把公司职员这个类的元素（即职员）进行分类，定义程序员类，同时也定义不同岗位的业务员类。如果说业务部的职员小李也会编写程序，那究竟该把小李放入程序员类还是业务员类呢？

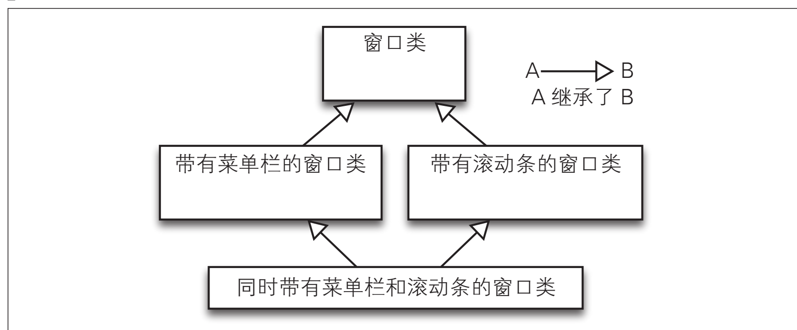
一个人同时承担程序员和业务员的角色的情况是完全有可能的。那

么一个类同时作为多个类的子类的情况就是很自然的了。也就是说现实世界中一种事物有可能属于多种分类。为了实现对这种现实情况的模拟，作为工具的程序设计语言是不是应该支持对多个类的继承呢？这就是多重继承的初衷。

多重继承对于实现方式再利用非常便利

多重继承对于实现方式再利用是一种非常便利的方法。图 12.6 展示的就是多重继承的一个例子。带有菜单栏的窗口这种类继承了窗口类并且实现了菜单栏的操作。另外，带有滚动栏的窗口类也继承了窗口类，并且实现了能滚动内容的滚动栏的功能。如果要实现一种既具有菜单栏又具有滚动栏的窗口该怎么做呢？在支持多重继承的语言中，只需要继承两个类就可以实现。

图 12.6 GUI 和多重继承



另外，作为实际使用的代码的一个例子，我们来看从 Python 语言标准库 SocketServer.py 中取出的四行代码。

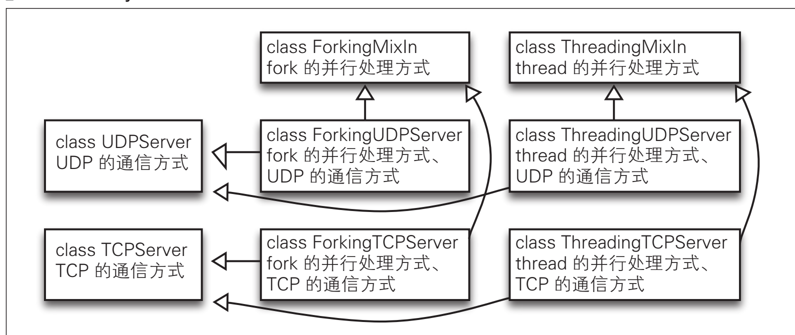
Python标准库中使用的多重继承示例

```
class ForkingUDPServer(ForkingMixIn, UDPServer): pass
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
```

这段代码定义了四个子类，用来表示在通信中使用 UDP 还是 TCP 以及并行处理中使用 fork 还是 thread。这四个子类是由两个选项结合两

个类做多重继承定义的（图 12.7）。pass 一词的意思是没有特殊的处理。比如定义 ForkingUDPServer 只需要继承 ForkingMixIn 和 UDPServer 就可以完成，不需要其他的代码。

图 12.7　Python 标准库中使用的多重继承

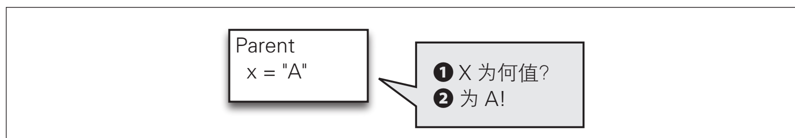


12.3
多重继承的问题——还是有冲突

多重继承看起来真的很方便。但是，使用多重继承时该如何解决名字解释的问题呢？当问到类中 x 值是什么时，该如何回答呢？

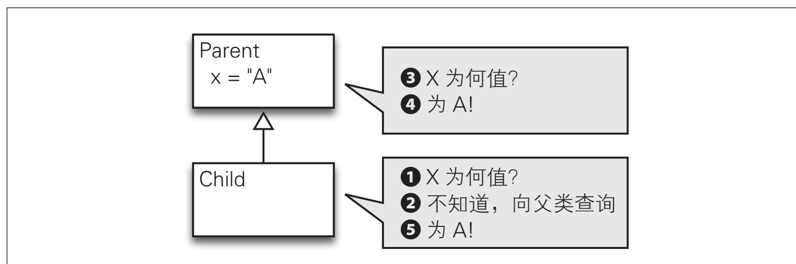
首先，如果这个类本身知道答案，就直接给出回答（图 12.8）。

图 12.8　名字解释之 1



其次，如果这个类本身不知道答案，就去问它的父类再给出回答（图 12.9）。

图 12.9　名字解释之 2



那么，像下面的代码中展现的多个父类具有相同名字的方法时会怎么样呢?

Python
```
class ParentA:
    x = "A"

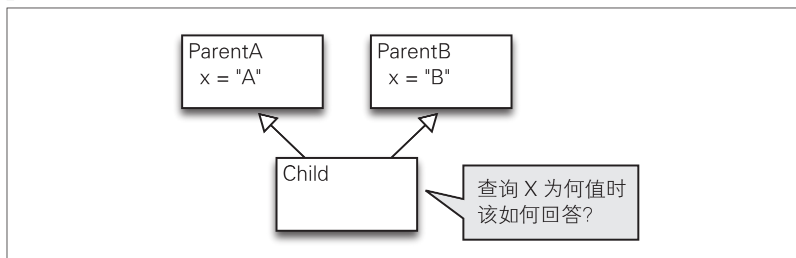
class ParentB:
    x = "B"

class Child(ParentA, ParentB):
    pass

print Child.x  # 正确输出是什么?
```

究竟应该调用哪个方法呢?这里再一次出现了名字解释的问题（图 12.10）。

名字解释之 3



解决方法 1：禁止多重继承

Java 语言中就禁止了类的多重继承。只要不认可类的多重继承这种方式，就不会有上述问题。这样可以把问题解决得很干脆，只是会以失去多重继承的良好便利性为代价。

除此之外，在 Java 语言及其相关库中也可以观察到舍弃作为实现方式再利用的继承的倾向。比如图形工具套装的实现就是这样。1995 年发布的 Abstract Window Toolkit（AWT）中，它是通过继承对各种方法进行重载的。而现在在 Eclipse 等语言中使用的 Standard Widget Toolkit（SWT）却不允许再使用继承。

委托

取而代之发展起来的概念是委托[①]。这种方法定义了具有待使用实现方式的类的对象，然后根据需要使用该对象来处理。使用继承后，从类型到命名空间都会被一起继承，从而导致问题的发生，这种方法只是停留在使用对象的层面上。

下面一段代码显示了一个使用委托的例子。

```java
public class TestDelegate {
    public static void main(String[] args){
        new UseInheritance().useHello(); // -> hello!
        new UseDelegate().useHello();    // -> hello!
    }
}


class Hello{    ❶
    public void hello(){
        System.out.println("hello!");
    }
}
```

① 委托（delegation）也叫做聚集（aggregation）或者咨询（consultation）。

```
class UseInheritance extends Hello {      ❷
    public void useHello(){
        hello();                          ❸
    }
}

class UseDelegate {                       ❹
    Hello h = new Hello();                ❺
    public void useHello(){
        h.hello();                        ❻
    }
}
```

显示“Hello !”的方法 hello 为类 Hello 所持有（❶）。类 UseInheritance 通过继承类 Hello 自身也持有了方法 hello（❷）并加以使用（❸）。与之不同，类 UseDelegate 并没有继承类 Hello（❹），而是通过句❺持有了类 Hello 的对象。当有需要使用时通过句❻将需要的处理委托给该对象操作。

与从多个类中继承实现强耦合的方式相比，使用委托进行耦合的方式显然要更好一些。对于委托的使用，也不需要在源代码中写死，而是可以通过配置文件在合适的时候注入运行时中去。这个想法催生了依赖注入（Dependency Injection）的概念。

接口

刚刚提到 Java 语言中禁止了多重继承，但它也具备实现多重继承的功能。这就需要借助接口（interface）[①]。

接口是没有实现方式的类。它的功能仅仅在于说明继承了该接口的类必须持有某某名字的方法。多重继承中发生的问题是多种实现方式相冲突时选取哪个的问题。而在接口的多重继承中，尽管有多个持有某某方法的信息存在，但这仅仅表明持有某某方法，不会造成任何困扰。下面一段代码中就继承了持有相同名字的方法的两个类，编译时不会发生

① Java 语言中类的继承用 extends，接口的继承用 implements 来区别表示。另外接口的继承也称为实现。

错误[1]。

```java
public class TestMultiImpl implements Foo, Bar {
    public void hello(){
        System.out.println("hello!");
    }
}

interface Foo {
    public void hello();
}

interface Bar {
    public void hello();
}
```

这段代码中，类 TestMultiImpl 继承了 Foo 和 Bar 两个接口。如果这个类中不实现 public void hello ()，编译时将出现“没有实现应该实现的方法”这样的错误[2]。也就是说，继承了接口 Foo 后，这个类就作为一种类型表现出必须持有 public void hello () 的特点，可以让编译器对它进行类型检查。

Java 语言为了仅实现功能上的多重继承引入了接口。PHP 语言和 Java 语言一样不认可多重继承，并从 2004 年发布的 PHP5 开始引入了接口的概念。

解决方法 2：按顺序进行搜索

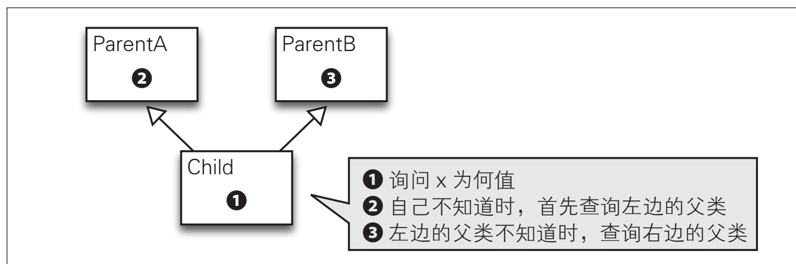
曾经也有些语言试图通过明确定义搜索顺序来解决冲突问题。但是该如何定义呢？如果单纯定义说当前类回答不了时就去检查首先书写的（左边的）类，按这样的顺序（深度优先搜索法）可行吗？（图 12.11）

---

① Java 语言中可以对名字相同但参数类型不同的方法进行重载。因此，准确来讲，这里说的名字应该是签名。

② 具体来讲，提示的错误消息应该是“TestMultiImpl 不是 abstract 的，Foo 中的 abstract 方法 hello( ) 没有被重载”。

图 12.11　从左边开始搜索的顺序



深度优先搜索法的问题

遗憾的是这种方法可能造成不自然的结果。请看下面的代码。

```python
class Parent:
    x = "A"

class Child(Parent):
    x = "B"

print Child.x   #-> 重载之后变成B

class Base:
    x = "A"

class Derived1(Base):
    pass

class Derived2(Base):
    x = "B"

class Multi(Derived1, Derived2):
    pass

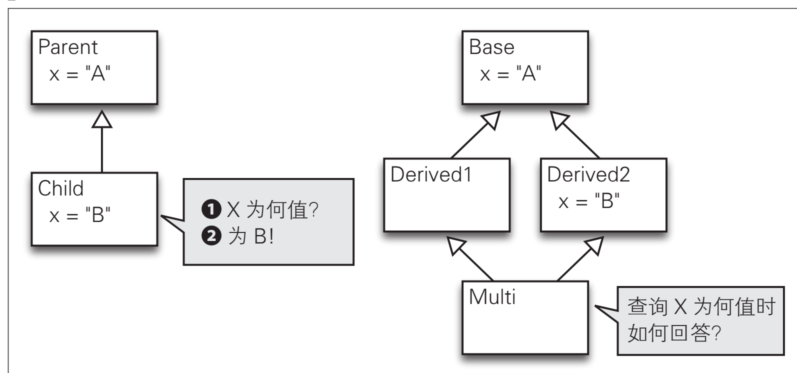
print Multi.x   #-> 该输出何值?
```

从类 Base 继承了两个类 Derived1 和 Derived2。又有一个类 Multi 从这两个类继承出来。这样产生的继承关系叫菱形继承。

方法是可以被重载（override）的。图 12.12 的左边的 Parent 中定义

的 x 在 Child 中就被重载了。因此 Child 中的 x 值变成了重载后的值。那么图 12.12 右边的菱形继承中 Derived2 方法重载之后，Multi.x 的值应该是多少呢？

图 12.12 重载和菱形继承



Multi 首先检查左边的父类 Derived1，Derived1 再检查它的父类 Base，结果是 A。Python 2.1（2001 年）就是遵循这种方式的（深度优先搜索法）。然而使用这种方法的话，初始的 x 和重载后的 x 混合后的结果仍然是初始的 x。这样一来，好不容易重新定义的 x 丢失了。

为了回避这一问题，从 Python 2.3（2003 年）开始使用 C3 线性化[①]。

C3 线性化确定顺序

C3 线性化是于 1996 年提出来的一种算法[②]，它对类进行编号以满足以下两个约束条件：

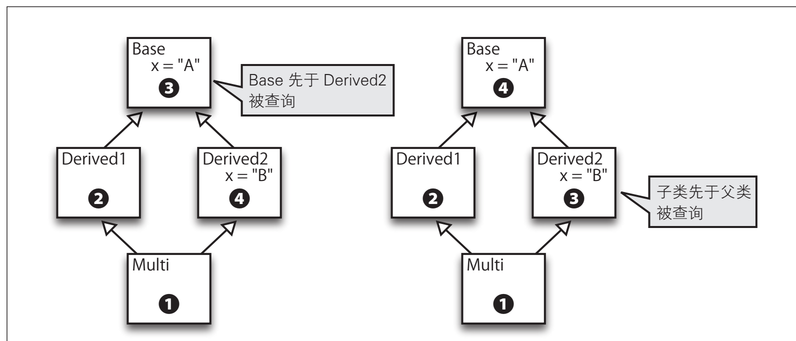
- 父类不比子类先被检查
- 如果是从多个类中继承下来则优先检查先书写的类

① 2001 年发布的 Python 2.2 中引入了一种新风格的类（new-style class），同时决定方法搜索顺序的算法也得到了改良。但是，文档上的理论和实际实现之间有种种差异，几经波折之后，Python 2.3 还是确定采用 C3 线性化。

② 在 OOPSLA'96 的活动上，Kim Barrett 等人发表了题为“A Monotonic Superclass Linearization for Dylan”的演讲。

之所以出现重载后的值变回初始值，就是因为父类先于子类被检查。因此，在此有针对性地加上了第一个结束条件（图 12.13）。

图 12.13　左：深度优先的搜索顺序 右：C3 线性化的顺序



下面的代码反映了使用深度优先搜索法的 Python 2.1 和采用了 C3 线性化顺序的 Python 2.3 以后的版本之间程序行为的差别。

```
Python 2.1
# 尽管在Derived2中对 x 进行了重新定义
# 到Multi后又回到了原来的情况
class Base:
    x = 'A'

class Derived1(Base): pass

class Derived2(Base):
    x = 'B' # 重新定义

class Multi(Derived1, Derived2): pass # Derived1在左边

print Multi.x #-> 'A'
# ↑没有使用Derived2中的定义

Python 2.3以降
# new-style class（继承自object，使用C3线性化）
# Derived2中定义的'B'被Multi继承下来了
class Base(object):
    x = 'A'
```

```
class Derived1(Base): pass

class Derived2(Base):
    x = 'B' # 重新定义

class Multi(Derived1, Derived2): pass

print Multi.x #->'B'
# ↑使用了Derived2中的定义
```

Perl 语言曾经允许在库的层面交替使用深度优先搜索法、广度优先搜索法[①]和 C3 线性化。从 Perl 6 开始已采用 C3 线性化的方法作为默认的程序行为。

解决方法 3：混入式处理

原本，问题是指从一个类到它的祖先类[②]有多种追溯方法。既然如此，定义仅包含所需功能的类并把它与需要添加这些功能的更大的类糅合在一起不就行了吗？我们把这种设计方针、混入式处理方式和用来混入的小的类统称为混入处理（Mix-in）。据说[③]Mix-in 一词起源于 C++ 语言设计者斯特劳斯特卢普经常光顾的 MIT（麻省理工学院）附近的一家甜品屋，指的是把坚果、葡萄干和冰淇淋混合在一起[④]。

菱形继承在使用了 Mix-in 之后就可以变形为非菱形继承。图 12.14 的左侧展示的是类 AB 从类 A 继承并添加了方法 B，类 AC 从类 C 继承并添加了方法 C，进而从类 AB 和类 AC 做多重继承得到了类 ABC。右侧展示的是通过从类 A 和做混入处理用的类 B、类 C 按实际需要继承得到类 AB、类 AC、类 ABC。这种方法不仅消除了菱形继承，而且继承

① 简单来说，这是一种先将所有直接父类穷尽再去检查父类的父类的搜索方法。

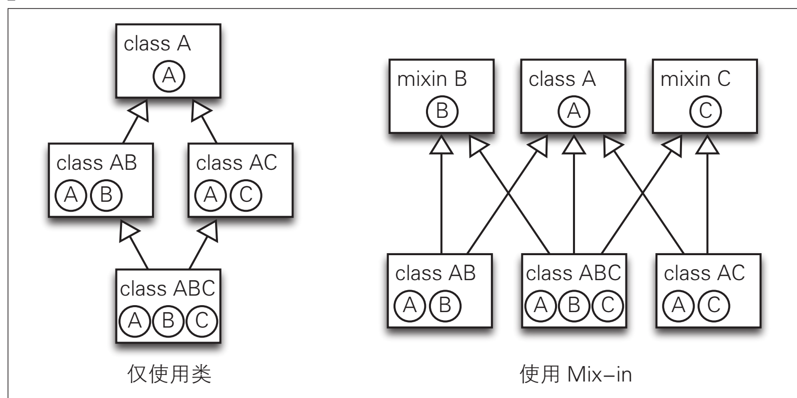
② 祖先类是包括了父类以及父类的父类等的统称。

③ 请参考斯特劳斯特卢普《C++ 语言的设计与演化》，http://www.stroustrup.com/dne.html。

④ 笔者也曾去过那家甜品屋吃过冰淇淋，能按自己独特的需求配制出心仪的口味的冰淇淋是件挺有趣的事情。

树的深度也减少了一层。

图 12.14 左：菱形继承 右：使用 Mix-in 消除菱形继承



Python 语言中

Mix-in 这种编程风格本身并不依赖于语言处理器，即使不被支持也可以使用。在 12.2 节 SocketServer.py 的多重继承这个例子中，ForkingMixIn 和 ThreadingMixIn 就是为实现混入式多重继承的小型的类。通常这些小型的类最小限度地定义了一些方法，起到了作为代码再利用单位的作用。然而它们却不用于单独创建实例。为了表明这一点，Python 语言会在该类的名字中加上 MixIn 来标识。

Ruby 语言中

Ruby 语言采用的规则是：类是单一继承的而模块则可以任意数量地做混入式处理。模块无法创建实例，但可以像类一样拥有成员变量和方法。也就是说，模块实质上是从类中去除了实例创建功能。即使类的多重继承被禁止了，通过使用模块的 Mix-In 方式照样可以实现对实现方式的再利用[①]。

下面的代码中，模块 Hello 中定义了方法 hello，模块 Bye 中定义了方法 bye，类 Greeting 是这两个类混入式处理的结果。最后两行代码的

① Mix-in 并不能解决所有的名字冲突的问题。图 12.14 右侧的类 A 和 mixin B 或者 mixin B 和 mixin C 有相同名字的方法时，仍然会导致冲突的产生。

运行结果证实了 Greeting 的对象同时具有方法 hello 和方法 bye。

```ruby
module Hello
  def hello
    puts "hello!"
  end
end

module Bye
  def bye
    puts "bye!"
  end
end

class Greeting
  include Hello
  include Bye
end

Greeting.new.hello  #-> hello!
Greeting.new.bye    #-> bye!
```

解决方法 4：Trait

多重继承在一开始是如何导致问题发生的？发表于 2002 年的一篇关于 Trait 的论文[1]很好地整理了其中的问题点。

类具有两种截然相反的作用。一种是用于创建实例的作用，它要求类是全面的、包含所有必需的内容的、大的类。另一种是作为再利用单元的作用，它要求类是按功能分的、没有多余内容的、小的类。

当类用于创建实例时，作为再利用单元来说就显得太大了。既然如此，如果把再利用单元的作用特别化，设定一些更小的结构（特性 = 方法的组合）是不是可以呢？这就是 Trait 的初衷。这和类另外去定义再

① Nathanael Schaerli, Stephane Ducasse, Oscar Nierstrasz, Andrew Black, “Traits: Composable Units of Behaviour”, 2002.

利用单元的方法不同，和 Ruby 语言中的模块很相似。那么它们之间有什么差别呢？

名字冲突时的程序行为

我们首先来看 Ruby 语言的问题点。

```
Ruby 1.9.3
module Foo
  def hello
    puts "foo!"
  end
end

module Bar
  def hello
    puts "bar!"
  end
end

class Foobar
  include Foo
  include Bar
end

class Barfoo
  include Bar
  include Foo
end

Foobar.new.hello   #-> bar!
Barfoo.new.hello   #-> foo!
```

在这段代码中，类 Foobar 中 include 了模块 Foo 和模块 Bar。但是这两个模块都拥有方法 hello，于是产生了冲突。Ruby1.9.3 规定发生类似的冲突时，默认选取最后被 include 的类 Bar 中的方法 hello。这样一来，程序员就会忽略冲突的存在。一旦 include 的顺序变成类 Barfoo 中那样，程序行为又不一样了。

Trait 的设计使得即使顺序改变了程序的行为也不会变。发生名字冲

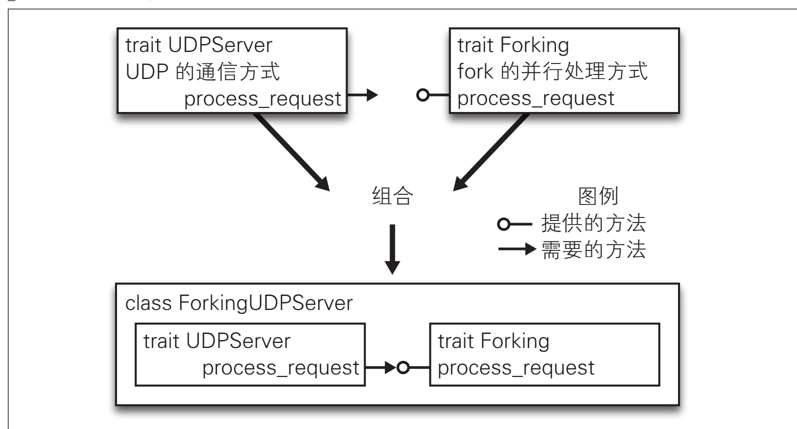
突时，程序会明确地发布错误信息。作为 Smalltalk 语言处理器之一的 Squeak 就提供了为方法取别名和指定不参与冲突的方法等冲突的解决方法。

提供的方法和所需的方法

实际上前面章节提到的 Python 语言中 Mix-In 的例子里也存在问题。在 Python 语言的标准库中，将 ForkingMixIn 混入类 UDPServer 中创建了一个新的类。ForkingMixIn 属于不能创建实例的类。那 UDPServer 是能够创建实例的类么？比如，如果没有 ForkingMixIn 提供的方法 process_request，程序还能执行么？如果是那种非混入式地提供方法 process_request 不可的类，那么"可以单独用来创建实例"这一说法就多少显得有些误导了。

Python 的问题在于缺少一种手段来表明类在什么状态下是可以创建实例的。Scala 语言的 Trait 技术中提供了方法来声明何种方法为必要的。如图 12.15，Trait 提供的方法用圆圈、Trait 需要的方法用箭头表示。UDPServer Trait 需要方法 process_request，同时 Forking Trait 提供了方法 process_request。通过两者的组合得到了可以创建实例的类 ForkingUDPServer。

图 12.15　提供的方法和需要的方法



其他各功能

另外，可以利用已有的 Trait，通过改写某些方法定义新的 Trait 实现继承。还可以通过组合多个 Trait 实现新的 Trait。这就是 Trait 的概要说明。它一方面把问题妥当地分而治之，一方面又因为功能繁多令人困惑。读者们想必都还记得 goto 语句就是因为其功能过于强大而退出历史的舞台的吧。所以说力量过于强大未必是件好事。

Trait 逐渐被广泛采纳

Trait 最早是在 Squeak 语言中引入的。同样使用了 Trait 的还有 Scala 语言。但是或许是因为这两种语言对类型的态度的差别，其中 Trait 的实现方式各有繁简。Perl 6 也引入了和 Trait 相似的功能，称为 Roll。PHP 语言也从 5.4 版本开始引入这种功能。Ruby 语言则在 2.0 版本中引入了 mix method，可以像 Trait 一样操作模块[①]。

12.4 小结

有这么多的解决方法，哪个才是最好的呢？这因具体情况而异。我们了解了 Trait 这种方法的提出以及逐渐被很多语言采纳的过程，它看起来是一种很有前途的方法。

不可否认，现在看起来是最佳的解决方案将来难保还是最佳。或许 10 年 20 年后会被更好的解决方案取代，像动态作用域一样被淘汰出局。

然而，笔者认为，Trait 技术是一个很好的开端。它认为类同时具有的作为再利用单元和实例生成器的两种作用是相反的。或许类这一概念作为面向对象的根基具有不可动摇的地位。然而这一概念本身也是从一个雏形慢慢发展得越来越复杂，进一步整理之后再逐渐让渡出某些功能的。现在备受关注的 Trait 和一些其他概念也必将不断地演变下去。经过长时间琢磨沉淀，一部分将臻于成熟被推广使用，最后将变成现在的

① RubyConf 2010 Keynote(2), http://www.rubyist.net/~matz/20101113.html.

静态作用域和 while 语句那样被认为是理所当然的存在。

专栏

从头开始逐章手抄

针对面对庞大信息量心力交瘁时该怎么办的问题，我们在第 6 章“学习讲求细嚼慢咽”专栏中介绍了三种方法。第一种方法是“从需要的地方开始阅读”，第二种方法是“先掌握概要再阅读细节”（第 8 章）。当这两种方法都不奏效时，就要采用最后一招——从头开始逐章手抄。

当没有明确要做的事情或者想要了解的东西时，当简单浏览的内容过目即忘时，以这种学习状态，不管怎么学也无法获得真知。

因此，为了打基础抄写教科书吧。不学习光唉声叹气是徒劳无用的，不如什么也不多想估且先做知识的搬运工。

除此之外别无它法。笔者有一种比较喜欢的方法，按时间段来衡量学习效果，比如每隔 25 分钟看自己学到了多少。当然按学习量来衡量也是可以的。重要的是设定足以获得成就感的合适的学习间隔。

后记

由于篇幅所限，本书只论述了程序设计语言的一些重点话题。希望读者在阅读本书后能获得一个整体性认识。

如果你读过本书之后，对某些话题的产生了兴趣，想要了解更详细的内容，那是最好不过。兴趣提升学习效率，兴之所至，才能学得深、学得透。在学习了详细内容之后，回过头来再看这一整体框架就更容易理解了。

接下来，我就对本书未能尽述的内容做个简略的补充。

本书提到一个问题会有多种解决方法，或许有读者想知道究竟哪种方法是最好的。那些敢做出断言的宗教人士或占卜师似乎向来人气很高，然而，现实中的大部分问题都需要具体问题具体分析，很难讲哪种最好。最佳设计方案也会因要求的不同而加以修正。针对假设的特定情况进行的速度等的最优化，有可能影响某种方案在其他情况下的应用。关于该话题，读者可以关注富豪式程序设计和 YAGNI 等关键字 ①。

可以说，只要是编程，大概都不可避免要产生一些错误（bug）。为了使程序准确运行，就需要调试（debug）。错误是由程序解释和现实情况不一致引起的。要尽早发现错误的存在，可以使用 assertion、test 等表达程序应有行为的解释性语句。要尽早地发现错误产生的原因，可以去学习调试的方法论和调试工具的使用方法。基于 Jenkins 等平台的持续集成（continous integration）可以频繁进行自动软件测试，从而能尽早发现存在的问题。

现在的编程不再单一枯燥。借助编辑器可以实现语法着色，借助 Eclipse 等集成开发环境可以在后台进行编译，并在有问题的代码下面用波浪线标示。这些功能让编程工作变得丰富多彩。但还有人想使编程变

① 富豪式程序设计是日本图形用户界面设计专家增井俊之倡导的编程风格。它主要针对用户界面开发，主张为了实现简单直观的界面可以不惜代码方面的投入。与此成鲜明对比的是 YAGNI（You aren't gonna need it，你不会需要它）的编程思想，它主张程序员不应该添加任何当前认为不需要的功能。——译者注

得更加轻松，于是出现了 Emacs 等一些编辑器的代码自动补充功能、Snippet 插件和 Eclipse 中的 Quick Fix 这样能大大减轻代码输入压力的功能。而对于一些语言来讲，某些集成开发环境提供的代码重构（code refactoring）支持也是非常有益的。

如果你在一个团队中从事开发工作，那就必然会关心源代码的版本管理和代码可读性的问题。这在单人编程工作中同样需要考虑。因为写完程序一个月后你可能就不再是当初的你，早已忘记了那些当初没有在代码中体现出来的编程的前提事项。另外，不单对自己的程序，还有必要去好好理解他人编写的程序。

我们很容易更加关注 How 的问题（即如何去实现），实际上，What（要实现什么）和 Why（为什么要实现）的问题也是不可忽略的。How 充其量只不过是手段。然而时间是有限的，干劲是宝贵的，要把它们用在真正需要的地方。凡此种种，书不尽言，篇幅所限，至此搁笔。

版权声明

Programming

Language

Concepts

多角度剖析编程核心概念 | 掌握编程语言共通的知识

- 为什么编程语言中会有函数、类、作用域等概念?
- 为什么语言设计者会设计出这样的语法?
- 本书从编程语言设计的角度，对多门编程语言进行比较，深入剖析了编程语言中核心概念产生的原因和使用方法，对理解编程语言的本质大有裨益。

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/编程语言与程序设计

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-36153-0

定价：45.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果有关电子书的建议或问题，请联系专用客服邮箱ebook@turingbook.com。

在这里可以找到我们：

微博@图灵教育：好书、活动每日播报

微博@图灵社区：电子书和好文章的消息

微博@图灵新知：图灵教育的科普小组

微信图灵访谈：ituring_interview，讲述码农精彩人生

微信图灵教育：turingbooks